An Orthogonal Framework for Fault Tolerance Composition in Software Systems

Sobia Khurshid Khan

Computing Department

Lancaster University

United Kingdom

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

OCTOBER 2015

Acknowledgments

I would like to start my acknowledgments by thanking Almighty Allah for giving me strength to complete this thesis. I would extremely thankful to my supervisor, Dr Lynne Blair, for all the guidance and comments provided in all this thesis work. The support she provided had a great influence to my experience and in the success of this thesis completion. She has been model of patient and support. I am also thankful to my supervisor Prof. Awais Rashid for his continuous support and guidance.

I am grateful to all the members of room C31 for the great work environment. They have provided me opportunities for fruitful discussion and feedback on my research.

I want to express my sincere thanks to my parents specially my dad for his continues prayers and moral support. I am extremely grateful for my husband Zafar, who has been constantly caring and supportive in all things over the years. I also thank my wonderful kids: Muhammad and Saliha for always making me smile.

I thank my entire family member for all the support, belief and encouragement, which had a great importance in my personal commitment to this thesis. I want to mention the name of Weronika Khan for her support she gave me with my kids over the years. During all my work, she never said ‘No’ to me whenever I was in need of her help. Thank you very much!

Abstract

Building reliable systems is one of the major challenges faced by software developers as society is becoming more dependent on software systems. The failure of any system can lead to a serious loss, for example serious injury or death in case of safety critical systems and significant financial loss in the case of business-critical systems. As a consequence, fault tolerance is considered as a solution to provide reliability, but the fault tolerance capability is associated with many challenges, such as the right development phase where it needs to be introduced, how it can be composed with the software, and the issues that arise from this composition such as complexity and potential undesirable feature interactions.

This thesis presents an orthogonal fault tolerance framework for the composition of design diversity fault tolerance mechanism with the base system. It further ensures the separation of concerns between the 'base' system and the fault tolerance mechanisms that are composed with the base system. The composition in this framework is based on operational semantics that describe the behaviour of the underlying components when composed with the fault tolerance mechanisms. A custom-built pre-processor is based on these composition rules, and is used to automatically compose the system component and the fault tolerance mechanisms. The very introduction of different fault tolerance mechanisms to the system may cause interactions with other fault tolerance features or with system components. Logic properties written in CTL and LTL are used in NuSMV to analyse undesirable interactions.

To illustrate its applicability, the framework has been applied to the Home Automation and Therac-25 software.

Declaration

This thesis has been written by myself, and the work reported herein is my own. Many of the ideas in this thesis were the product of discussions with my supervisors Prof. Awais Rashid and Dr Lynne Blair. The work reported in this thesis has not been previously submitted for a degree in this, or any other form.

Sobia K Khan

Table of Contents

Acknowledgements i

Abstract ii

Declaration iii

Table of Contents iv

List of Tables ix

List of Figures x

Chapter 1: Introduction

1.1 Introduction to Fault Tolerance.. .. 1

1.2 Software Fault Tolerance... 2

1.2.1 Data Diversity and Design Diversity Techniques..................... 2

1.2.2 Single Version Software Fault Tolerance Mechanisms............. 3

1.2.3 Multiple Version Software Fault Tolerance Mechanisms....…. 4

1.2.4 Software Fault Tolerance at Requirement Specification and Design Levels.. 6

1.3 Software Fault Tolerance and Separation of Concerns (SoC)........... 6

1.3.1 Aspect Oriented Software Development................................ 7

1.3.2 Orthogonal Variability Model (OVM)..................................... 7

1.3.3 The Potential for Interaction of Fault Tolerance Features...... 7

1.4 Research Issues, Aims and Objectives... .. 8

1.5 Novel Contributions... .. 9

1.7 Thesis Outline.. .. 10

Chapter 2: Background Work

2.1 Introduction... … 11

2.2 Fault Tolerance at Requirement Specification and Design Level 13

2.2.1 Co-operative Architectural Style... 13

2.2.2 iFTComponent (idealized Fault Tolerant Component).............. 15
2.2.3 Architectural Patterns.. 16
2.2.4 MAL Specification 18
2.2.5 Aereal Framework (ACME Specification) 19
2.2.6 DRIP Catalyst 21
2.2.7 Event B (DEPLOY Project) 22
2.2.8 Other Approaches 23
2.2.9 Summary 24
2.3 Aspect-Oriented Modelling and Design Approaches....................... 27
2.3.1 Aspect-Oriented Architecture Model (AAM) 27
2.3.2 Theme 28
2.3.3 Motorola WEAVER.. 29
2.3.4 AspectJ for Exception Handling.. 31
2.3.5 Other Approaches 32
2.3.6 Summary 33
2.4 Feature Interaction Analysis.. 35
2.4.1 FI Filtering 35
2.4.2 FIN Method (Use Case Driven Analysis)................................... 36
2.4.3 CHISEL 37
2.4.4 Other Approaches 38
2.4.5 Summary 40
2.5 Comparison and Discussion.. 40

Chapter 3: Orthogonal Fault Tolerance (OFT) Framework

3.1 Introduction.. 45
3.2 Orthogonal Fault Tolerance Model... 47
3.2.1 Feature Dependency Analysis.. 48
3.2.2 Constraints... 49
3.2.3 Typical Design Diversity Mechanisms...................................... 50
3.3 Composition in the Orthogonal Fault Tolerance (OFT) Framework...... 51
3.3.1 Introduction to LTS (Labelled Transition Systems).................... 51
3.3.2 Operational semantics... 51

3.3.3 Conditions and Actions over Variables.................................... 54
3.3.4 Handling 'Generics' for Fault Tolerance Components.............. 56
3.3.5 Component Composition with Fault Tolerance Conditions...... 60
3.3.6 Pre-Processor Tool (Lex & Yacc)... 60
3.3.7 Revisiting other Fault Tolerance Features handled by 'Generics'. 64
3.4 Feature Interaction Analysis.. ... 66
3.4.1 Categories of Feature Interaction... 67
3.4.2 Specification of Properties in CTL/LTL..................................... 68
3.4.3 Reasoning about Feature Interactions through CTL/LTL and Model Checking... 70
3.5 Summary... … 72

Chapter 4: Illustrative Case Study: A Home Automation System

4.1 Introduction..… 73
4.2 Formalisms/Operational Semantics.. 75
4.2.1 Applying Labelled Transition Systems to the Home Automation System... 75
4.2.2 Operational semantics for the composition of Home Automation components.. 76
4.3 Introducing Fault Tolerance into the Home Automation System...... 77
4.3.1 Applying an Acceptance Test Fault tolerance formalism......... 77
4.4.2 Applying a Fault tolerance formalism for Parallel Execution – with Voter.. 79
4.4.3 Applying the Fault tolerance formalism for the Hybrid Voter 82
4.4.4 Pre-Processor Tool (Lex & Yacc)... 83
4.4.5 Lex and Yacc generated NuSMV Models for Home Automation Components... 85
4.5 Feature Interaction Analysis.. .. 87
4.6 Introducing New Features in Home Automation........................... 89
4.7 Summary...… 90

Chapter 5: Case Study: Therac-25

5.1 Introduction... .. 92

5.1.1 Accidents in Therac-25... 93

5.1.2 Recommendations... 94

5.1.3 Orthogonal Fault Tolerance and the Therac-25 System........... 94

5.1.4 Features of the Therac-25 System... 95

5.2 Formal Specification of the Therac-25 System............................ … 96

5.2.1 Statechart Model for the Therac-25 Components.................... 97

5.2.2 Finite State Machine Model for the Therac-25 Components….. 99

5.2.3 Summary.. 101

5.3 Fault Tolerance Composition.. …. 101

5.3.1 Composition of N-version Programming Mechanisms with the Therac-25 Component... 102

5.3.2 Pre-Processor Tool (Lex & Yacc).. 105

5.4 Model Checking for the Safe Condition and Feature Interaction Analysis..…. 106

5.5 Summary.. …. 109

Chapter 6: Evaluation and Analysis

6.1 Introduction.. …. 110

6.2 Hypotheses... …. 111

6.3 Analysis... …. 112

6.4 Further Analysis.. …. 115

6.5 Conclusion.. …. 117

Chapter 7: Conclusions

7.1 Introduction.. ….. 119

7.2 Summary of the thesis.. ….. 119

7.3 Contribution of the thesis.. ….. 120

7.4 Future Work... ….. 123

7.5 Concluding Remarks.. …. 124

References 125

Appendix A 134

Appendix B 146

List of Tables

Table 2.1: Requirement specifications and design approaches............ 26

Table 2.2: Aspect Oriented Modelling and Design Approaches............ 34

Table 2.3: Feature Interaction Analysis….. 40

Table 2.3: Overall Comparison Table.. 44

TABLE 3.1: Fault tolerance mechanisms features................................ 47

Table 6.1: Soundness with respect to interaction detection.............. 115

Table 6.2: Overall Evaluation.. 117

List of Figures

Figure 3.1:	Context of the Proposed Method...	46
Figure 3.2:	Orthogonal Fault Tolerance Feature Model................................	49
Figure 3.3:	A state machine representing an Acceptance Test mechanism and checkpointing...	57
Figure 3.4:	Workflow of Proposed Feature Interaction Analysis Approach...	66
Figure 3.5:	Model Checking Based Feature Interaction Analysis.......	72
Figure 4.1:	A Feature Diagram for a Smart Home.......................................	73
Figure 4.2a:	Light Controller (LC) State Machine..............................	74
Figure 4.2b:	Home Status Controller (HSC) State Machine............................	74
Figure 4.3:	Composed State Machine for LC and HSC..................................	75
Figure 4.4:	A state machine representing Acceptance Test mechanism and checkpointing...	77
Figure 4.5:	A state machine representing voting mechanism with backward error recovery......................................	80
Figure 4.6:	A state machine representing voting mechanism with forward error recovery..	80
Figure 4.7a:	Light Controller (LC) with AT State Machine..............................	85
Figure 4.7b:	Home Status Controller (HSC) with AT State Machine.................	85
Figure 4.8:	Composed Fault Tolerant Home Automation State Machine………	86
Figure 4.9:	Potential Feature Interactions...	87
Figure 4.10:	Sequential Action Trigger...	89
Figure 4.11:	Climate Controller triggers the Alarm Component, Light Controller and Home Status Controller...	90
Figure 4.12:	Shared trigger interaction between Climate Controller, Alarm and Light Controller..	90
Figure 5.1:	Therac-25 Software Features..	96
Figure 5.2:	Therac-25 Interface Statechart [Bolton et al. 2008]....................	97
Figure 5.3:	Therac-25 Machine Component Statechart...............................	98
Figure 5.4:	Therac-25 Interface Component..	99
Figure 5.5:	The Machine Component of the Therac-25……..........................	100

Figure 5.6: Component to compute sensor values for the spreader position with N-version Programming…………………………………………………………………… 103
Figure 5.7: A state machine representing the result of voting mechanism composed with Interface component finite state machine……………………………….. 106
Figure 5.8 Interface Component with voting result for Spreader position……….. 106

Chapter 1

Introduction

1.1 Introduction to Fault Tolerance

Building reliable systems is one of the major challenges faced by software developers as society is becoming more dependent on software systems. Reliability of a system is one of the attributes of dependability along with availability, safety and security [Lapri 1985]. The failure of any system can lead to a serious loss, for example serious injury or death in case of safety critical systems and significant financial loss in the case of business-critical systems. However, even though a system may be viewed as being safe (in the sense that no-one has knowingly been injured), the system may still not be reliable. Faults may exist in the software that have not yet escalated into accidents.

For this thesis, it is the reliability of systems' software that is of primary importance, rather than safety. Fault tolerance and fault avoidance can be seen as constituent attributes of reliability. The field of fault tolerant systems is a mature one, dating back to the 1960s, and concerns the design and development of systems that have the capability (without external assistance) to continue to operate correctly even in the presence of faults [Avizienis 1976]. Providing fault tolerance capability at the implementation level is the traditional way for achieving reliability and can be time and cost effective. However, modern research has also focused on providing fault tolerance at the requirement specification and design levels, and formal properties relating to reliability can be guaranteed with the help of verification and model checking support.

Various fault tolerance mechanisms have been developed that lie mainly in two major groups: hardware fault tolerance and software fault tolerance [Chen and Avizienis 1978]. Hardware fault tolerance mechanisms deal with *physical faults* in hardware components whereas software fault tolerance mechanisms deal with *software design faults* or *programming errors*. Hardware fault tolerance can be provided by techniques that introduce

redundancy through the replication of components alongside a voting mechanism, for example triple modular redundancy [Avizienis 1971]. Typically, components have multiple physical backups and are separated into small parts where extra redundancy is built into all physical connectors, power supplies, etc.

Software fault tolerance is provided by techniques such as data diversity and design diversity [Chou 1997]. In *data diversity*, a software system's input data is changed (re-expressed) and/or broken down into smaller units and then later re-combined [Amman and Knight 1987]. In contrast, *design diversity* proposes the use of different versions of the same software system [Avizienis and Chen 1977]. It is *software* fault tolerance that this thesis will focus on.

In general, the early efforts in building fault tolerant systems were focused mainly at the later stages of software development such as the implementation level and integration and testing level. However, dealing with fault tolerance concerns in the early phases of software development can help developers better manage software risks through the early identification and resolution of errors and faults [Romonovsky 2007]. Many efforts have already been made to provide fault tolerance at the levels of requirement specification and design, e.g. providing fault tolerance at the architectural level, by using ADLs or formal methods, as well by using formal or semi-formal modelling techniques at the design level. Fault tolerance has also been considered as a 'separate concern' and handled, for example, in aspect-oriented approaches.

The position of this thesis is to consider software fault tolerance at the *design level* in order to lessen the effort and cost to recover from faults and failure at the later stages of software development. Furthermore, the thesis focuses on providing fault tolerance as a separate concern that is *orthogonal* to the underlying system behaviour. This separation of fault tolerance concerns ensures that fault tolerance features can be introduced independently, are re-usable and can be more easily maintained. However, this separation implies the need for subsequent *composition* of the separated components and fault tolerance concerns.

1.2 Software Fault Tolerance

As mentioned above, there are many mature software fault tolerance mechanisms that are based on data diversity and design diversity techniques.

1.2.1 Data Diversity and Design Diversity Techniques

Data diversity approaches use only one version of software but different data spaces though logically equivalent yet diverse sets of input [Amman and Knight 1987]. The N-copy programming approach is the main example of this technique in which each copy of versions runs in parallel with different sets of input data. An example of data diversity is a minor perturbation in input values to get a new set of input values; these new values may avert software failure and hence allow accurate results to be produced.

On the other hand, *design diversity* techniques use different versions of the software rather than the data. These versions are not copies of each other; instead they are independently designed with different design teams and algorithms – all written to the same specification [Avizienis and Chen 1977]. An example of design diversity is N-version programming where independently written, yet functionally equivalent, versions of software that execute in parallel. A majority voting algorithm then compares all results, and decides (according to its algorithm) which result or results are correct.

This thesis will focus on *design diversity* techniques for software fault tolerance.

In addition to these two styles of diversity, software fault tolerance mechanisms can be divided mainly into two further groups; *single version* software fault tolerance mechanisms (that are used for various techniques including data diversity), and *multiple version* fault tolerance mechanisms that are used for design diversity.

1.2.2 Single Version Software Fault Tolerance Mechanisms

In addition to considering data diversity, single version software fault tolerance mechanisms include checkpoint and restart mechanisms, process pairs, atomic actions and exception handling [Torres 2000].

Checkpoints and restart mechanisms are significant effort towards fault tolerance and recovery. Although this is considered primarily as a single version fault tolerance technique, it is also used in multiple version techniques like Recovery Block (see below). In this technique, states of the system are saved periodically and, on the occurrence of a system failure, this saved state is used to restore the service to the previous checkpoint, and the processing can resume from that point. After the successful completion of the tasks, these checkpoints are deleted to save the memory space [Pradhan 1996].

A process pair is also a single version technique. This technique uses two identical components that run concurrently on different processors where one is the primary

processor and the other is the secondary (or backup) processor. In the case of a failure, control from the primary processor passes to the secondary processor. In this technique, program modules are used to decompose a problem into independent components. These components are designed to have built in protection to keep any abnormal component behaviour in one module from affecting the other modules. Beside fault tolerance, additional benefits of using modularisation include testing and easier maintenance [Pradhan 1996].

An atomic action comprises of a group of processes, where these processes can interact with each other but not with the rest of the system. In the case of a failure, an exception is raised and all of the processes in the atomic action can invoke an exception handler to handle that failure; this means that the processes have the ability of self checking. Hence the component has the ability to detect certain errors and can take some steps to prevent them from spreading all over the system. The error detection mechanisms include many checks like replication, timing, coding and structural checks [Anderson and Lee 1981].

Exception handling techniques are used to detect, contain and recover from unexpected or exceptional conditions that are contrary to the system's normal behaviour. Some examples of exceptional conditions include wrong input data or data corruption, invalid service requests or system design faults [Randell 1975].

1.2.3 Multiple Version Software Fault Tolerance Mechanisms

Multiple version software fault tolerance mechanisms use multiple versions (or variants) of software based on design diversity strategies to provide fault free processing. In design diversity strategies, design of the software is typically written by different design teams to follow the same specification. These versions can be executed either in parallel or sequentially. The main motivation behind the different versions is the expectation that components built differently should fail differently as they have different designers, different algorithms, and different design tools. In case of the failure of one version, at least one of the alternative versions should be able to continue processing to produce the correct result. Examples of these mechanisms include recovery block, N-version programming and N-self checking programming.

The basic **Recovery Block** mechanism was first developed by Horning et al. in 1974 [Horning et al. 1974] and was implemented by Randell in 1975. Recovery block mechanisms use acceptance tests and a backward recovery scheme, and also need to save the state of the

variant currently being executed. In this mechanism, different variants are run *sequentially* such that, after one version has been run, its output is passed through an acceptance test (AT). If the acceptance test is passed then the result of that variant is considered as the overall output. If the acceptance test fails, the system should use checkpoints and retrieve its saved safe state or error-free state to restart the execution from that state but using the next variant [Randell 1975].

N-Version Programming (NVP) was suggested by Elmendorf in 1972 [Elmendorf 1972] and developed by Avizienis and Chen in 1977. This mechanism typically uses a *parallel* execution scheme in which the different variants are executed in parallel and the different outputs are passed to the adjudicator. The adjudicator may use voting or consensus voting to select the appropriate result. Alternatively, in this mechanism a sequential execution scheme can be used along with checkpoints to save the state of the primary variant before an alternate variant is executed [Avizienis 1985] [Hecht 1979].

N-Self Checking Programming (NSCP) was developed by Laprie et al. in 1988. This mechanism uses program redundancy to check its own behaviour during execution. The adjudicator in this case can be the combination of both the acceptance test and the voter, referred to as a hybrid voter. In other words, this mechanism itself is the combination of recovery block and N-version programming. In this mechanism, the execution scheme for different versions can be sequential or parallel and each version uses a separate acceptance test. As above, if sequential execution is used, checkpoints are needed to store the state for each version [Laprie 1990].

The Consensus Recovery Block (CRB) mechanism was developed by Scott in 1983, also combining N-version programming and recovery blocks [Scott 1983]. This mechanism uses n independent versions of a program, a decision algorithm similar to N-version programming and an acceptance test similar to that used in a recovery block mechanism. All versions are executed in parallel and submit their results to a voter. If two or more versions agree on the same result, the result is considered as a correct output. If there is no output, or the versions produce wrong or multiple outputs then an acceptance test is used. The result of the primary version is first passed through the acceptance test. If it passes, it is considered as a correct output. If the output fails, the result of the next version is passed to the acceptance test. This process continues until the correct result is found.

This thesis will focus on *multiple version* techniques for software fault tolerance.

1.2.4 Software Fault Tolerance at Requirement Specification and Design Levels

Over the years, major causes of faults in software systems have been investigated, including faults introduced due to bad requirement specification and design. Under the premise that it is more time and cost effective to recover from such faults at these earlier stages of software development, many experiments have been performed to test the effectiveness of various design diversity and multiple version fault tolerance mechanisms; for example, see [Dahl and Lahti 1979] and [Kelly and Avizienis 1983].

Approaches addressing fault tolerance at the requirement specification and design levels include architectural approaches with or without formal language specification, modelling techniques such as labelled transition systems, and model checking techniques to verify fault tolerance properties.

In summary, this thesis addresses the provision of software fault tolerance mechanisms at the *requirement specification and design level*, and will focus on using design diversity and multiple version techniques.

1.3 Software Fault Tolerance and Separation of Concerns (SoC)

Separation of Concerns (SoC) is considered as a concept for decomposing a software system into small, loosely-coupled parts called concerns, to deal with the complexity of software system [Dijkstra 1982]. Each of these concerns is implemented using modularisation mechanisms, and hence is developed, updated and maintained independent to each other and can be reused.

Fault tolerance in a software system has the nature of a crosscutting concern, as it has a global impact on a system and its functionality is scattered over the system in several components. Like safety and security, fault tolerance concerns are non-functional aspects of the system and, according to the underlying principles of a separation of concerns, can be separated from the functional aspects of the system.

In addition, separation of fault tolerance concerns does not need to be considered only at later stage of software development; rather, it is beneficial to consider it at early stages of software development such as requirement specification and design levels.

1.3.1 Aspect-Oriented Software Development

Aspect-oriented software development is a broad research area that offers various new modularisation mechanisms for software systems in order to separate out multiple cross-cutting concerns from the underlying base system. A small number of aspect-oriented software development approaches have addressed the issue of fault tolerance by separating fault tolerance concerns from the base system. [France et al. 2004] proposed an aspect-oriented modelling approach that allows developers to conceptualize, describe and communicate logical dependability solutions in isolation, with the help of an aspect-oriented architecture model (AAM). An alternative aspect-oriented modelling approach is proposed by [Clarke and Walker 2001][Clarke and Walker 2002] where a design, called a Subject, is created for each system requirement and presented as UML model views.

1.3.2 Orthogonal Variability Model (OVM)

The idea of orthogonality in this thesis has been inspired by OVM, the Orthogonal Variability Model [Pohl et al. 2006]. The key benefits of OVM are the improvement of decision making, communication and traceability. It also aims to make the overall development process simple, consistent and unambiguous [Pohl et al. 2006].

Design diversity fault tolerance mechanisms consist of different mandatory and optional features such as the selection of an adjudicator or the number of variants, etc. These features have dependency relationships between each other [Patrick et al. 2009]. Orthogonality, as a separation of concerns mechanism, can help a designer in understanding the impact of dependency relationships and decision making in selecting an appropriate fault tolerance feature since fault tolerance features are isolated.

In this thesis, the incorporation of design diversity fault tolerance mechanisms as an *orthogonal* concern is considered to bring the same benefits as mentioned above, including reusability, modifiability, and reducing cohesion and dependability between a system's components and fault tolerance mechanisms.

1.3.3 The Potential for Interaction of Fault Tolerance Features

The separation of fault tolerance concerns from the underlying system components, and the subsequent *composition* stage, brings an increased difficulty in system comprehension as it introduces new interdependencies and interrelationships between fault tolerance features

and software components. If separate fault tolerance concerns are integrated without consideration of their relationship, dependency and impact on other components, it can potentially lead to undesirable and unwanted *feature interaction* problems.

1.4 Research Issues, Aims and Objectives

The overall goal of the thesis is to present an orthogonal fault tolerance approach for designing fault tolerant systems, composing the orthogonal components and analysing potential undesirable interactions arising from this composition. The primary purpose of this orthogonality is to separate the fault tolerance concern from the main software system to make the overall system consistent, simple and symmetrical. The orthogonal fault tolerance approach has the following objectives:

- To provide a separation of fault tolerance concerns from the underlying components of the software system. These separate elements must be effective when composed with the components of the existing system. Hence, a composition mechanism provides the way to compose fault tolerance features with the components of the existing system. This separation of concerns is expressed with an orthogonal view of the system and ensures that the adaptation of fault tolerance features can be reasoned about independently before their composition with the components of the existing software system.
- To explicitly deal with design diversity and multiple version software fault tolerance mechanisms, including recovery block, N-version programming and N-self checking programming mechanisms.
- To support reasoning about different fault tolerance features as well as system components before composition takes place in terms of their dependency relationships and constraints.
- To deal with any potential undesirable feature interactions that arise through the composition of fault tolerance features with the components of the software system. In the case of such scenarios, model checking can be used to check for feature interactions and, following this, the same style of fault tolerance features can be applied reflexively to resolve undesirable interactions.

The proposed approach is supported by the model-checking tool NuSMV and also a custom-built pre-processor written in Lex & Yacc. This pre-processor automatically generates the composition of the orthogonal fault tolerance components, and also serves to provide the

input translation for the model-checking tool. CTL can be used to verify properties of the composed system in conjunction with NuSMV to analyse potentially undesirable feature interactions.

Finally, the approach will be evaluated with the help of two case studies, namely a smart home automation system and the Therac-25 computer-controlled medical machine for radiation therapy. In Therac-25, the difference between reliability and safety becomes apparent. The generation of early Therac machines, ones preceding the Therac-25, were deemed to be safe even though some of the same software defects existed: safety threats were averted by a hardware interlock system. Hence, when the failures occurred they did not escalate into accidents such as injury or death, i.e. the system was safe but not reliable.

1.5 Novel Contributions

In addressing the objectives outlined above, this thesis will make the following novel contributions:

- Orthogonal Fault Tolerance as a Separation of Concern

  It is the first work to propose an orthogonal fault tolerance model as a means to manage and reason about multiple design diversity fault tolerance mechanisms that may need to co-exist in a software system. The proposed Orthogonal Fault Tolerance (OFT) framework provides design diversity fault tolerance mechanisms and can be reasoned based on feature hierarchy, dependency and constraints. It provides a separation of fault tolerance concerns to study the effect of new or modified fault tolerance features on the system without adding complexity to the base system.

- Fault Tolerance Composition

  The proposed Orthogonal Fault Tolerance framework approach uses labelled transition systems and operational semantics to formally underpin the composition approach. The approach also introduces the concept of fault-tolerance 'generics' for specifying different fault tolerance mechanisms.

- Deals with potential Fault Tolerance Interactions

  The proposed approach explicitly deals with the problem of any potential undesirable feature interactions arising from composition of new/updated fault tolerance features with the component of software system. A model checking approach is used to analyse

these interactions and a reflexive approach that uses a similar style of fault tolerance is proposed for the resolution of these interactions.

1.6 Thesis Outline

Chapter 2: Introduces and discusses the related work from the literature in detail. In particular, work related to architectural fault tolerance, formal methods approaches to composition, aspect-oriented development considering separation of cross-cutting concerns and feature interaction handling techniques are considered.

Chapter 3: An overview of the proposed Orthogonal Fault Tolerance (OFT) framework approach is presented in chapter 3 with detailed methodology and supporting tools. This chapter outlines the formalisms used in the composition of fault tolerance with the system and the methodology to deal with the interactions.

Chapter 4: This chapter illustrates the methodology and formalisms with the worked case study of a home automation system. This chapter further illustrates how undesirable feature interactions may arise from the composition of fault tolerance with the components of underlying Home Automation system.

Chapter 5: In this chapter, the methodology is applied to the historically significant case study of the Therac-25 computer controlled medical machine for radiotherapy. This case study demonstrates the use of the proposed approach to compose orthogonal fault-tolerance concerns with the Therac-25 system, in order to demonstrate the removal of well-known and well-documented software errors.

Chapter 6: This chapter presents an analysis and discussion regarding the evaluation of the proposed methodology having been applied to the case studies. From this, conclusions are drawn about the methodology soundness, effectiveness and efficiency. This chapter offers a critical analysis of the whole presented work.

Chapter 7: This chapter concludes by returning to the aims and objectives outlined in chapter 1 and evaluating to what extent they have been achieved. The shortcomings, future recommendations and expansions of the proposed approach are also presented in this chapter.

Chapter 2

Background Work

2.1 Introduction

As already mentioned in Chapter 1, society is becoming more and more dependent on software systems and correspondingly a strong emphasis is on a high assurance that these systems are indeed reliable. As a consequence, fault tolerance is considered as a solution to provide reliability, but the fault tolerance capability is associated with many challenges, such as the right development phase where it needs to be introduced, how it can be composed with the software, and the issues that arise from this composition such as complexity and potential undesirable feature interactions. This chapter outlines the research in support of introducing fault tolerance mechanisms at the requirement specification and design levels, and dealing with composition and feature interactions. It also identifies the gap in the state of the art and a research agenda, which introduces the framework for the fault tolerance composition and analysis of feature interactions. In order to make a comparison between different approaches, surveyed approaches are broken down into the following areas:

- **Fault Tolerance at Requirement Specification and Design Levels**: These approaches address fault tolerance at the initial phases of software development, such as specification, design and architecture levels.
- **Fault tolerance Composition**: These approaches have composition specific mechanisms to integrate different components of the software system.
- **Feature Interaction Analysis**: These approaches have the capability to analyse and detect undesirable feature interactions.

In order to make a comparison between different approaches, the focus will be on the following desired key characteristics as discussed below:

System Model: This characteristic shows the type of model being used to express the design of the system. Examples in terms of modelling are UML, statecharts, finite state machines, formal methods, temporal logic and UML sequence diagrams.

Fault Tolerance Expressiveness: This characteristic shows what types of fault tolerance mechanism are supported by the approach and what kind of faults can be handled. Examples are design diversity or data diversity approaches, and different faults like design errors.

Composition Mechanism: This characteristic describes the framework used to compose the fault tolerance mechanism with the underlying base system.

Separation of Concerns: Another important characteristic is the separation of fault tolerance concerns. Different approaches address this separation differently, for example by addressing it at a semantic or syntax level, or in case of fault tolerance by differentiating between normal and abnormal behaviour. Some approaches address this separation by designing functional components and non-functional features separately.

Feature Interaction Analysis: According to this characteristic, after composing fault tolerance mechanisms with the system's components, there is a need to analyse potential undesirable feature interaction that may lead the system into an inconsistent or error state.

Platform and Paradigm dependencies: This characteristic focuses on any platform and paradigm constraints used by the different approaches. These constraints can be programming languages like C, Java, C++, etc. or can be operating systems like UNIX, GNU/Linux, Windows, etc.

Supporting tools and examples/ case studies: This characteristic considers the supportive tool(s) used by each approach. Furthermore, it briefly describes the examples and case studies used to demonstrate the approach.

Firstly, each approach is presented with the specified characteristics presented above. Following this, a comparison will be made based on the characteristics that match the challenges mentioned in chapter 1, particularly fault tolerance expressiveness, orthogonality/separation of concerns, composition and feature interaction analysis. Section 2.5 then reviews an overall comparison and analysis of the underlying approaches that are related to the work in this thesis.

2.2 Fault Tolerance at Requirement Specification and Design Levels

The approaches listed in this section have a common characteristic: providing fault tolerance support at the design and requirement specification levels, rather than at implementation and testing levels. Each of the main approaches will be discussed in detail, but a summary of notable other approaches will also be included at the end of each section.

2.2.1 Co-operative Architectural Style

The approach in [deLemos 2001] introduces a co-operative architectural style for modelling and analysing fault tolerant software systems built from commercial off-the-shelf (COTS) components, considered as a black box. The co-operative architectural style offers the means to structure the complex applications and provides a way to add exception handling fault tolerance mechanism to untrustworthy components. In a co-operative style of architecture, the abstraction is provided in the form of 'connectors' that capture the collaborative behaviour between the architectural components. Components embody computation, whereas, connectors are mediating interactions between the architectural components. These connectors are used to limit the impact of the change to the overall system architecture. In this approach, system exceptions are handled at the component level, providing dependability from the untrustworthy components. The approach uses the UPPAAL model checker and timed automata to analyse the normal and abnormal behaviour of the co-operative architecture.

Similar to the [deLemos 2001] approach, [Issarny and Banatre 2001] present the implementation of exception handling within the components and connectors and at the level of architectural configuration. Components and connectors can raise exceptions. Exception handling within components and connectors handles exceptions internally to the specific component or connector and has no impact on the rest of the architecture.

System Model: These approaches are architecture centric, where components and connectors are used to describe normal and exceptional (abnormal) behaviour of the component. Components support the representation of structural and behavioural aspects of a system. Structural behaviour of component is described by name, attributes, description of structure such as composed-of, and intra-relations between different components. Similarly, the behaviour aspect of the component identifies the port of the component, normal, exceptional and failure behaviour of the component. Co-operative connectors

encapsulate the collaborative activity between the several components. The behaviour of the connectors use pre-conditions and post-conditions to start and finish a certain activity.

Fault Tolerance Expressiveness: These approaches deal with the exception handling that uses a forward error recovery mechanism to bring the system into a new or error-free state. For the illustration of exceptional behaviour, a handler defines its start and finish event based on pre and post conditions. It also deals with two types of failure behaviours: failure of omission and the failure of commission. The exceptional behaviour of a connector is represented with timed automata with data variables and based on the pre and post conditions of the collaborative operations.

Separation of Concerns: Co-operative style connectors are used to separate the normal and abnormal behaviour, providing a level of separation of fault tolerance concerns. Exceptional behaviour of connectors is represented by timed automata with data variables. The exceptional signal is raised with the occurrence of the exceptional behaviour when the pre or post condition for the collaborative operation is not true. This invokes the execution of the exception handler, and the co-operation is finished assuming that the post-condition for exception behaviour is true.

Composition Mechanism: The incorporation of an exception handling fault tolerance mechanism can be done with the co-operative architectural styles using components and connectors. The architectural connectors are used to add or change the behaviour of untrustworthy COTS components to build the dependable component. In this approach, connectors are considered as first class entities that describe the collaborative behaviour, which provides the basis for implementing error recovery in the presence of faults.

Feature Interaction Analysis: The interactions between the components are dealt with through the help of co-operative style connectors by identifying the normal, exceptional and failure behaviours. This approach however, does not address the feature interactions between the components specifically, and does not deal with undesirable interactions. In this approach, in fact the interactions between different components are carried out with the help of connectors to overcome the problem of architectural mismatch as COTS components are used for building the dependable components. Also, the interactions are explicitly addressed at a configuration level rather than at the component level. In a co-operative style architecture, specialised connectors are used to capture the coordinated behaviour of components.

Platform/paradigm constraint: These approaches are not platform or paradigm dependent.

Tool Support/Case Studies: The deLemos's approach uses UPPAAL model checker [UPPAAL] and timed automata with extended data variables. The approach is demonstrated on Self Gas service and VS-40X sounding rocket case studies [deLemos 2001]. Whereas, Issarny and Banatre's approach uses the Aster framework [Issarny and Banatre 2001] to provide an implementation for configuration level exception handling.

2.2.2 iFTC (idealized Fault Tolerant Component)

[Guerra et al. 2002] present the C2 style architecture for an idealized fault tolerant component architecture model (IFTCM), separating normal and abnormal activity and introducing specialised C2 connectors. The iC2C style [deLemos 2001] is used to produce an iCOTS protective wrapping for the components and provides an application specific fault tolerance capability. A similar approach is presented in [Brito et al. 2009] where architectural abstractions are used for building fault tolerant element (iFTElement), and can be seen as an extended form of idealized Fault Tolerant Components (iFTC). These elements can be instantiated as a component (iFTComponent) or a connector (iFTConnector) to carry out the computation and coordination respectively.

This approach deals with exception handling based on the methodology presented in [Rubira 2005] and [Brito et al. 2005]. These approaches define the exception behaviour, and separate the normal and abnormal behaviours using UML sequence and activity diagrams at the architectural level. The process algebra CSP is used to give the semantics of UML sequence diagrams, whereas the B-method is used for the semantics of UML entities.

System Model: In this approach Use Cases and UML component diagrams are used to describe scenario and system structures. Normal and abnormal behaviour of the component is dealt with the UML sequence diagrams and UML activity diagrams. The B method and CSP are used to define the semantics of the components' behaviour. Two main things are dealt with through these patterns: first the component implements crash-failure semantics, and secondly it claims to support dynamic reconfiguration.

Fault Tolerance Expressiveness: This approach deals with fault detection and then a fault's localisation and removal. The communication between idealized fault-tolerant components is only through request/response messages. The response can either be normal, or otherwise a 'failure exception' in the case of an invalid service request or due to a failure in

processing a valid request. Internal exceptions are associated with errors detected within a component that may be corrected, allowing the operation to be completed successfully; otherwise, they are propagated as external exceptions

Separation of Concerns: As idealized fault tolerant components separate the normal and abnormal behaviour, they can be seen as a separation of concerns.

Composition Mechanism: The composition of fault tolerance in a component based system is carried out with the help of the C2 architectural style. Request/ response is the only way of communication between components. An idealized fault tolerant component encapsulates the normal and abnormal behaviour inside it. Furthermore, idealised C2 components can be integrated into any C2 configurations with this C2 architectural style. This composition provides loose coupling between components as these components may be completely unaware of each other, especially when one integrates various commercial off-the-shelf components (COTS), which may have heterogeneous style and implementation language.

Feature Interaction Analysis: Idealised fault tolerant components are integrated together with C2 style architecture, thus allowing the interaction of iC2Cs with other idealized and/or regular C2 components. This interaction is for the communication between different components. There is no mechanism to analyse feature interaction.

Platform/paradigm constraint: As the approach works at the design and modelling level, there is no discussion about platform dependency. However, this approach targets critical component base systems and service oriented legacy architectures.

Tool Support/Case Studies: Both the B-method [Abrial 1996] and CSP [Agarwala and Tanic 1989] underpin the use of UML [Alhir 1998]. In extended approaches, ProB [ProB] and other testing tools have been used. A mine pump control system is used as a case study that can be found in [Mine].

2.2.3 Architectural Patterns

The work of [Parchas 2004] demonstrates that the application-independent fault tolerance techniques can be integrated in the architectures via a pattern. This approach is mainly used for web based services. The services are developed independently by different service providers and composed using patterns. A comparator provides semantics for similarity or dissimilarity of different services. A pattern for improving web services availability and

dynamic reconfiguration is proposed. It also addresses the dynamic composition of web services.

This approach deals with the fault tolerant redundant components for self checking pairs and architecture reconfiguration. The composition mechanism for incorporation of fault tolerance is via a pattern. The approach is based on diversity of different sources of information, comparator and bridges.

System Model: This approach uses architectural patterns, where the main elements are: bridge, comparator, service broker and FT-registry [Parchas 2004]. These architectural patterns are used for the elimination of mismatches between the required and provided services, and the provision and management of redundancies. These architectural patterns are described in terms of UML diagrams representing components, provided and required interfaces and connections between these interfaces. These patterns can tolerate failure by imposing crash-failure semantics when inconsistencies are detected in the data.

Fault Tolerance Expressiveness: This approach deals with the redundant components for self checking pairs, dealing with any kind of mismatch faults. An application-independent fault tolerant technique can be easily incorporated in architectural modelling such as multi-versioning derived from the notion of N-version programming (NVP). The architectural pattern in this approach resembles the N-self checking fault tolerance mechanism but does not employ two levels of comparison as it does not deal with the design faults. These patterns only deal with the structure of components, not their behaviour. Hence, these patterns are mainly suitable for the comparator element such as the self checking component that compares the mismatches and is assumed to tolerate single fault of the system. In case of checkpointing in backward recovery, these architectural patterns are not applicable as it requires integration with the functional behaviour of the system.

Separation of Concerns: The comparator provides the similarity and dissimilarity in the semantics of different data coming from distinct bridges. It monitors the behaviour of connected bridges and their internal behaviour. Although all architectural patterns are syntactically separated and specific to the certain element of the base model, they are not reusable. Moreover, with these architectural patterns, separation of concerns may not be directly applicable considering the fact that fault tolerance and fault intrusion are interrelated at an architectural level. However, these patterns are described in pattern languages that are not rigorously defined and therefore are not amenable as needed for

their identification associated with a particular style. Furthermore, there is no way to deal with dependability when using heterogeneous style of architecture patterns

Composition Mechanism: The incorporation of fault tolerance can be done with the architectural patterns by matching and mismatching patterns. There is no orthogonal fault tolerance rather it is embedded in the component. These patterns are also associated with the deployment constraints that are integrated with the specific system configuration.

Feature Interaction Analysis: There is no specific mechanism to analyse feature interaction in patterns for fault tolerant components.

Platform/paradigm constraint: The approach is not platform dependent but explicitly designed for the web based services.

Tool Support/Case Studies: The approach does not provide any tool support. Nor is it demonstrated on any case study.

2.2.4 MAL Specification

In particular, a small number of approaches use formal language specification for the development of fault tolerant systems. [Magee and Maibaum 2006] use the Modal Action Logic (MAL) specification language for specifying normal and abnormal states and behaviour of components. The component is described with its attributes, actions and axioms. The MAL specification is translated to finite state models with Labelled Transition System Analyser (LTSA) and Finite State Processes (FSP) algebra. Finite state models are used for automated verification of the required properties of fault tolerance.

The effectiveness of the fault tolerance models can also be verified by the finite state model derived from the MAL specification. The MAL specification is also used for the specification and reasoning about dynamically reconfigurable fault tolerance mechanisms for normal and abnormal behaviour, providing the specification for self healing and self checking fault tolerance mechanisms.

System Model: The specification used in this approach is component based. The key elements for fault tolerance representation include the description of component behaviour, interaction mechanisms such as connectors and coordination, configuration (with behaviour and states) and description of recovery activity. This approach uses finite state models, labelled transition system analyser and finite state process algebra for the specification. MAL

specifications use deontic operators (actions being obliged, permitted or forbidden). The semantics of these operators provide an explicit distinction between the ‘good’ or normal behaviour and abnormal or ‘bad’ behaviour. In the case of normative behaviour, good actions are executed and lead from good states to good states. In contrast, bad actions lead to the bad states where recovery is required to re-enter a normative behaviour mode. On the occurrence of bad behaviour, self healing recovery actions are specified.

Fault Tolerance Expressiveness: MAL specification deals with the self-checking/ self-healing fault tolerance mechanism with the help of fault tolerance models to demonstrate the effectiveness of the mechanism. The self healing property of the system is based on the ‘good’ and ‘bad’ states and their sub division to know: if nothing bad happens, then eventually normative behaviour is resumed. In some situations, normative behaviour is replaced by some appropriate fault tolerance model such as forward recovery. The Authors suggested further work to develop the reasoning to address the kind of reasoning for example new inference rules.

Separation of Concerns: The separation into good and bad states, and associated normative and abnormative behaviour, provides a level of separation of concerns between normal and abnormal behaviour. However, this approach does not provide separate fault tolerance models for the system recovery, as good and bad states co-exist within the component.

Composition Mechanism: There is no explicit definition of composition of fault tolerance with components of the system.

Feature Interaction Analysis: There is no explicit mechanism to analyse feature interaction in for fault tolerant components. However, the finite state model of the components of the system describing the good and bad states shows the normative and abnormal behaviour of the system.

Platform/paradigm constraint: The approach is neither platform dependent, nor designed for any particular paradigm.

Tool Support/Case Studies: The approach uses MAL specification and LTSA model checking tool to check the properties related to the fault tolerance. It is demonstrated on a simple example of master-server to slave-server.

2.2.5 Aereal Framework (ACME Specification)

The approach in [Filho, Brito and Rubira 2006] presents the Aereal (Architectural Exceptions Reasoning and Analysis) framework, with architectural descriptions of exceptions, and how exceptions flow between architectural elements. Architecture-based development with Aereal starts with the requirements analysis and architectural design of the system. The scenarios where system may fail or fault models are defined with the associated exceptions for each type of error and exception handling activity. These activities provide results that show a description of the system's architecture and their informal specifications. It further gives the fault model of the system and exceptional activity if any.

The approach integrates ACME, an architectural description language, for the specification of exceptional information, and uses Alloy, a first order relational language, to check the properties. The exceptional flows between different elements are captured in an Exception Flow Model which allows the specification of common rules of exception handling in software systems.

System Model: Exception flow between architectural elements is specified with the help of a software architecture view that depicts the exceptional components that catch or signal exceptions. These exceptional components are associated with the special connectors called exception ducts through which exceptions flow between components. Then the composition of the architectural description with the exception flow takes place, producing an extended architectural description. Finally the structural constraints are analysed to check whether the architectural description violates any of the exception constraints based on the Alloy specification.

Fault Tolerance Expressiveness: The special purpose architectural connectors are unidirectional links through which only exceptions flow. An exception flow adheres to one or more exceptional styles. Aereal includes an ACME specification for a basic architectural style for exceptions called SingleExceptionFam on which all exceptional styles are based to catch and/or signal exceptions.

Separation of Concerns: In the ACME specification, the architectural organisation and separation between normal views, normal architectural styles and normal connectors and exceptional views, exception styles and exception ducts promotes a clear separation of concerns at the architectural level.

Composition Mechanism: The composer tool provided by Aereal is used to read the ACME specification and the exception flow view and updates the former with exceptions defined by

the latter, generating an Alloy specification model. Exception flow views can be composed with the same architecture using the default semantic ‘union’.

Feature Interaction Analysis: There is no explicit mechanism to analyse feature interaction for fault tolerant components. However, the two separate design artefacts address the interactions within the architectural styles and the exception styles.

Platform/paradigm constraint: The approach is neither platform dependent, nor designed for any particular paradigm.

Tool Support/Case Studies: The approach uses the ALLOY model checker for exceptions and is demonstrated on a mining control system and a financial system case study.

2.2.6 DRIP Catalyst approach

In [Guelfi et al. 2004], an approach is presented that use MDE (Model Driven Engineering) process, UML based notation and MDA (Model Driven Architecture) concepts to support step-wise development of fault tolerant distributed systems, aiming to span design and implementation through generative methods. DRIP (Dependable Remote Interacting Process) is a framework specific MDE/MDA method, by combining model-driven generative and formal techniques. In this approach, Coordinated Atomic Actions (CAA) are used with the DRIP Catalyst development method to support backward and forward error recovery in distributed systems. It is called DRIP Catalyst, as it uses combination of MDE, UML-based notation, MDA tool and formal techniques.

System Model: This approach is architecture centric, and a UML-based language is used for the formal description of fault tolerance properties, using Java classes. CAAs (Coordinated Atomic Actions) are represented by UML activity diagrams that are very close to Java syntax; this allows the automatic transformation of the UML model to generated code. CAAs are built using the profile of specific design models. However, there is no general notation for platform-independent-design models.

Fault Tolerance Expressiveness: Coordinated atomic actions (CAAs) are used to address error recovery techniques and the combination of distributed transactions and atomic actions, and are applicable for both forward error recovery and backward error recovery. Forward error recovery relies on exception handling that transforms the system component to any correct state. UML based fault tolerant transaction profiles are designed for the

specific system that are syntactically and semantically well defined. The major disadvantage of these profiles is their specific dependency on platform.

Separation of Concerns: CAAs are designed as a set of roles cooperating inside each component and a set of resources accessed by them. As a result, each component has its own recovery scheme that allows the implementation of fault tolerance properties by raising and handling exceptions internally. Hence, this approach does not deal with fault tolerance as a separate concern.

Composition Mechanism: The incorporation of fault tolerance is done within the architectural Coordinated Atomic Actions. The CAAs are a unified scheme for supporting error recovery between interacting components. Hence, there is no orthogonal fault tolerance rather it is embedded in the component and is automatically transformed to the code level.

Feature Interaction Analysis: This approach does not verify any fault tolerance properties of their application, before generating the code. Hence, there is no assurance for the consistency between the verified model and the generated code. Moreover, there is no mechanism to analyse the inconsistencies and feature interactions in complex concurrent activities between different coordinated atomic actions.

Platform/paradigm constraint: The approach is explicitly Java platform dependent and was initially designed for object-oriented distributed systems and service-oriented applications.

Tool Support/Case Studies: The approach uses the CORRECT MDA tool [Guelfi et al. 2004] and the COALA language [Guelfi et al. 2004] as supportive tools.

2.2.7 Event B (DEPLOY Project)

In [Laibinis and Troubitsyna 2006], the authors propose a formal specification for patterns in a layered architecture for exception handling. This work also focuses on hardware faults and human errors. The B method is used to develop the layered architecture. This architecture-centric approach allows making a smooth transition from the architectural-level system modelling to specification and refinement of each particular component by using Event B. Event B is a state based formal approach and its formalism is considered as an extension of B method. The approach uses patterns to describe the module interface, which can be instantiated by component-specific data and behaviour during system refinement.

Moreover, the proposed refinement-based development techniques have coped well with modelling the complex mode transition scheme and verification of its correctness.

System Model: The textual Event B language is used for modelling purposes, describing a set of variables, their initial states, guarded transitions and other invariants. Modelling is also carried out with UML interface modules and graphical syntax of Event B. Event B also uses model checker ProB to verify the formalism. Rodin supportive tool can be used for the mathematical proofs.

Fault Tolerance Expressiveness: This approach does not deal with any specific fault tolerance mechanisms, nor does it handle any particular faults. However, dependability properties such as fault tolerance can be specified and then verified through theorem proving. In example of liquid handling workstation, error situations are anticipated with how to bring the system back into a normal state by either, roll forward, replicated processes or rollback as in checkpointing.

Separation of Concerns: This approach does not deal with fault tolerance as a separate concern or provide any fault semantics. There is no orthogonal fault tolerance rather it is separated by parameterisation and instantiation multiple times within the modules. The incorporation of fault tolerance can be done with the help of patterns in modules that split components into module interfaces and module bodies.

Composition Mechanism: The parallel composition with multiple assignments is used and satisfies the property of compositionality in the context of modules that describe as machines.

Feature Interaction Analysis: There is no explicit mechanism for identifying, detecting or analysing feature interactions between different components.

Platform/paradigm constraint: The approach can be used with ADA, C and Java and specialised for event based reactive systems.

Tool Support/Case Studies: The approach is applied on a liquid handling work station Fillwell. Atelier B automatic supportive tool for Event B is used [Fillwell™ 2002].

2.2.8 Other Approaches

Some other formal verification techniques have been proposed to address the fault tolerance in different software systems. In [Meng, Anita and Daniel 2009], the authors

analysed the effect of formal verification and testing due to adding different fault tolerance mechanisms to baseline systems. They concluded that re-execution was the most efficient mechanism, followed by parity code, dual modular redundancy and triple modular redundancy. They also presented the ratio of verification effort and testing effort to assist designers in their trade-off analysis when deciding how to allocate their budget between formal verification and testing and can be used in practical industrial production. [Yeung et al. 2003] proposed the CSP approach using FDR2 to formally verify both the fault-tolerant software and hardware design. Their work mostly focused on the implementation level using ProB and the JCSP programming environment [Welch and Martin 2000].

In [Tomoyuki et al. 2001], the authors proposed a symbolic model checking method using SMV with CTL [Clarke, Grumberg and Peled 1999] formula for the verification of fault tolerance of systems. [Lanfang et al. 2012] proposed a similar approach for formal verification of signature-monitoring mechanisms by model checking using operational semantics and model checking NuSMV [NuSMV]. In contrast, [Daniel and Ruben 2005] present the idea of using an aspect-oriented approach to add fault tolerance to software and then the formalism can be applied to the resulting separation of the fault tolerance code from the function code. [Clarke, Grumberg and Peled 1999] describe four kinds of fault tolerant abstraction, namely faults, fault-masking, voting and communications. These abstractions are then formalised and verified. [Kulkarni et al. 2005] propose a similar approach but mainly focus on the synthesis of fault tolerant system according to the algorithms which are mechanically verified by using PVS theorem [Owre et al. 1996].

The work of [Keinzle et al. 2005] presents a requirements engineering process to consider reliability and safety at early phases of software development process. This approach uses the concept of idealized fault tolerant components with the addition of Dependability Oriented Requirement Engineering Process (DREP). DREP extends traditional use case modelling to consider reliability and safety concerns.

The above approaches use formalisms for the automatic verification of fault tolerance properties at the initial phases of software development such as requirements specification and design level. The approach of Daniel and Ruben explicitly addresses the separation of fault tolerance concerns in terms of aspect oriented software development, whereas the MAL specification supports reasoning about the dynamic reconfiguration of the component and fault tolerance self-healing mechanisms.

2.2.9 Summary

All of the above approach deals with the fault tolerance at the requirement specification or design level mainly focusing on architecting fault tolerance. In these approaches, there is no clear mechanism to deal with the feature interaction or the composition of fault tolerance with the base system. Moreover, few approaches deal with the fault tolerance as a separate concern; more commonly it is embedded within the component.

Having presented a number of different approaches in detail above, a summary is presented in tabular form in Table 2.1. The table serves as a quick way to filter out the most desired properties such as fault tolerance expressiveness, separation of concern, composition and feature interaction analysis.

	Co-operative Architectural Style	Architectural Patterns	Mal Specification	Aereal Framework	iFTComponent	DRIP Catalyst	Event B
System Model	Components and Connectors	Bridges, comparator, service broker and FT registry	Finite state machine, LTS, and Process Algebra	Exception Styles Exception views Exception ducts	UML, CSP B Method	UML, CAA, MDE	UML, B method
Fault Tolerance Expressiveness	Exception Handling	Comparator acts as Self checking mechanism	Self checking and Exceptions	Exception Handling	Forward recovery (Exception Handling) /any kind of exceptions	Exception Handling	Not specific
Separation of Concerns	Partially Via Architectural styles	Comparator compared the matches and mismatches	Partially addressed With MAL specification	Partially addressed with normal and exceptional flow of information between components	Provides as computation is separated for component and connectors	Models	N/A
Composition Mechanism	Connectors	Via Matching Patterns	Views composition	Views composition with ACME specification	Within components and connectors	Within architectural coordination actions	With patterns and modules
Feature Interaction Analysis	Partially	Partially by matching similar and dissimilar patterns	N/A	Partially by separating normal and exception design artefacts	N/A	N/A	N/A
Platform/paradigm constraint	N/A	Web based services	N/A	N/A	Component based system. No platform dependent	Java Platform and object oriented development	ADA, C and Java Reactive systems
Tool Support/Case Studies	UPPAAL model checker, timed automata, Gas Auto filling system, VS-40X sounding rocket	N/A	MAL , LTSA Client Server System	Alloy, Mining Control system and Financial System	ProB, B method	MDE and COALA	Eclipse Rodin

Table 2.1 Requirement specification and design approaches

2.3 Aspect-Oriented Modelling and Design Approaches

There are many excellent survey papers on Aspect-Oriented techniques, including aspect-oriented analysis and design approaches [Chitchyan et al. 2005] and aspect-oriented modelling approaches [Schauerhuber 2007].

This section highlights some of the approaches that are particularly relevant given the thesis' focus on separating out fault tolerance concerns and composing such concerns with the base system.

2.3.1 Aspect-Oriented Architecture Model (AAM)

[France et al. 2004] proposed an aspect-oriented modelling (AOM) approach that allows developers to conceptualise, describe and communicate logical dependability solutions in isolation with the help of an aspect-oriented architecture model (AAM). This architecture consists of a set of aspect models and a base architecture. An integrated view of the architecture is obtained by composing the aspect and base architecture models to produce a composed AAM with the help of Templates and UML models such as class diagrams and sequence diagrams. Aspects describe solutions in this approach that crosscut UML module views and may specify concepts that are not present in a base model. The AAM approach provides support for the separation of crosscutting concerns such as dependability and also supports composition of aspect and model views using a composition strategy. Later, a tool was developed called Kompose [Fleurey et al. 2008] that uses the composition technique proposed by this approach. This approach is also useful in identifying and resolving conflicts that arise after the composition of aspects and models.

System Model: This approach uses Template, UML class diagrams, UML sequence diagrams and architectural views. In this approach, UML provides some support for multidimensional separation of concerns through the use of different diagram types that can be used to describe non-orthogonal views of a system.

Composition Mechanism: in this approach, aspect model is composed with the primary model based on the composition directives provided in AAM. Composition directives tell how the aspect model can be composed with the primary model. Logical view of architecture is presented by AAM, where UML notation and interaction diagrams are used for the primary model and dependability patterns are used for the aspect model. Composer tool has been built to automate this composition.

Feature Interaction Analysis: Interaction of concerns is addressed by this approach with the help of UML templates and patterns in Analyzer component of the AAM. The approach deals with the conflicts arising because of composing and integrating aspects and primary model views. Hence, it deals with the interaction and undesirable properties that arise as a result of integrating aspect and primary models. Similar to the composition procedure, the feature interaction analysis requires further effort on the part of the developers but the Model Analysis component is somehow responsible for analysing the composed model to identify errors and to determine the extent that dependability objectives are met.

Platform/paradigm constraint: The approach is not constrained to a particular platform except the use of UML.

Tool Support/Case Studies: A composition tool, Kompose/composer [Fleurey et al. 2008], has been developed that uses the same aspect oriented architecture model approach to compose aspects. A small case study of user management system is used to demonstrate the applicability of the approach.

2.3.2 Theme

The approach by [Baniassad and Clarke 2004] is one of the early aspect-oriented modelling approaches, and strongly supports work at the requirement specification and design levels. This approach represents crosscutting system concerns as features and aspects based on the requirement specifications. The approach is based on the composition patterns presented in [Clark et al. 2001], with the design unit named Theme. The base theme unit refers to the base system and the aspect theme refers to the crosscutting aspect as well as any other behaviour that is triggered by behaviour in some other theme. Modelling of the base theme is carried out in UML design process, whereas aspect themes are modelled with different modularisation packages. The modularisation of the aspect theme is similar to the UML models where the structural view is represented with the help of class diagrams and behavioural views is represented with the help of sequence diagrams.

System Model: At the requirement analysis phase, this approach uses Theme/Doc that provides views whereas, at the design phase, the approach uses Theme/UML: standard UML modelling with class diagrams using stereotypes, UML sequence diagrams to show the structural and behavioural view of the base theme unit, and the aspect theme unit. Action views in the requirements document are used to identify the crosscutting behaviours. Each action is designed separately with Theme/UML. Theme/UML is designed as a platform-

independent approach and provides mappings to AspectJ. Basically, Theme/UML poses no restrictions on what UML diagrams might be used for modelling. Moreover, particularly Theme/UML allows every concern to be refined separately and then to be composed into a new model.

Composition Mechanism: The Theme approach works at the two levels, at the requirement specification level and the design level. Composition patterns are used to compose concerns at the requirement and design level. In Theme/UML, first all non-crosscutting themes are composed and then crosscutting themes are woven one after the other into the composed model, thus forcing the developer to consider the ordering of crosscutting themes.

Feature Interaction Analysis: The UML class relationship is used to model interactions between different concerns, but these are not related to undesirable feature interaction in anyway.

Platform/paradigm constraint: The Theme approach is an aspect-oriented software development approach but not platform dependent.

Tool Support/Case Studies: The requirement views are demonstrated with the help of small example of Course Management Systems whereas the approach is applied on the case study of location aware game called the Crystal Game.

This approach does not deal specifically with fault tolerance mechanisms, but any fault tolerance mechanism can be seen in terms of aspects. The Theme approach not only deals with the aspect theme but any behaviour that can triggered by any other theme unit, hence, can be use for any kind of faults and exceptions raised.

2.3.3 Motorola WEAVER

In [Cottenier et al. 2007], a tool is developed known as Motorola WEAVER based on SDL, designed for UML Statecharts that include action semantics. It supports the automatic generation of source code and platform-specific models by weaving aspects into the executable UML models. Two main constructs are used by Motorola WEAVER: 'where' and 'what'. Where refers to the locations or jointpoints in the model where crosscutting behaviour emerges such as transitions. Special constructs known as pointcuts are used the jointpoint actions behaviours. 'What' refers to the behaviour of the crosscutting concerns.

An aspect encapsulates the multiple pointcuts and connectors and it also contains a binding diagram that defines which connectors are bound to which pointcut. The work of [Cottenier, Berg and Elrad 2006] describes composition semantics between aspects and base models by using metamodels. Crosscutting behaviour is designed with Specification and Description Language (SDL) state charts. There are two kinds of jointpoints, action jointpoints are used for the actions and calls construction whereas transition jointpoints are used for the tranitions in state machine. A connector is used for the implementation of state machine. The weaver engine semantically simulates and weaves the model. Moreover, all the weaving is done at the module level; there is no way to generate aspect code that is weaved together with the base code.

System Model: This approach uses UML Statecharts that include action semantics. Special constructs using stereotypes describe the pointcuts and connectors representing jointpoints and advice. Pointcuts and connectors are encapsulated by a special construct aspect. Finite state machines are also used to show the behaviour specifications of the components. A special <<bind>> stereotype defines which pointcuts needs to bind with which pointcut and similarly for the connectors.

Composition Mechanism: In the Motorola WEAVER, there are three categories for the composition mechanism: pointcut composition, connector composition and aspect composition. For the pointcut composition, Boolean operators (and, or, not) are used. In connector composition, the precedence relationship is considered and the <<follows>> stereotype is used to compose the connectors with joinpoints. Deployment diagrams are used for the aspect composition that binds the aspect with the base model.

Feature Interaction Analysis: The WEAVER approach explicitly specifies the precedence constraints for the pointcuts and connectors that can help reduce the undesirable interference at the same jointpoints. Moreover, dependability is also considered and the developer of one aspect must be aware of all potentially conflicting aspects, address conflicts between different features, hence analyse feature interaction particularly for the specific domain.

Platform/paradigm constraint: The WEAVER approach can be used with Java, C and C++ and platform dependent.

Tool Support/Case Studies: The Tau tool provides a model explorer that supports the detailed analysis of models written in C/C++. The approach is illustrated through two examples of resource server; transaction timeout aspect and two phase commit aspect.

2.3.4 AspectJ for Exception Handling

Several works also take an aspect-oriented approach to fault tolerance and exception handling concerns at the implementation level such as [Filho et al. 2005]. This approach works on the modularigin the exception handling with the help of AspectJ. AspectJ is considered as an extension to java based on aspect oriented concepts. The selected case studies are large, complete, deployable systems. All of the studies are evaluated against pre-defined attributes for example, cohesion, coupling, conciseness and separation of concerns. The study shows that AOP improves separation of concern between exception handling code and normal application code, also promoting the reuse of handlers.

This approach uses matrices to quantitatively measure the attributes such as separation of concerns. Three matrices are used for this purpose, namely concern diffusion over components that counts the number of classes and aspects, concern diffusion over operations that counts the number of methods, and advice and concern diffusion over lines of code that counts the number of transitions on each line of code.

System Model: In this approach, the error handling parts of the code such as 'try-catch', 'try-catch-finally', and 'try-finally' blocks of the Java code are modularised in aspects. These aspects are implemented with before, after and around advice, depending on the execution of the handler. Aspects to handle exceptions are defined for each class in the base system. The advice is expressed in terms of pointcuts and join points, which describes precisely where the additional behaviour should be added, e.g. before, after or around existing methods.

Composition Mechanism: Composition of aspect is carried out with the help of process called weaving in which java classes represent aspects. Weaver tool is used to perform weaving between aspects.

Feature Interaction Analysis: The approach does not address the issues of feature interaction or the interactions between exception handling aspects and aspects implementing other concerns.

Platform/paradigm constraint: The exception handling can be incorporated in many object oriented programming language such as Java, C# and C++ and normally considered as application specific.

Tool Support/Case Studies: Telestrada's Complaint Management Subsystem (CMS) is taken as a case study for this approach consisting of more than 12000 LOC and more than 300 classes.

This approach explicitly deals with exception handling fault tolerance mechanisms. In this approach, the error handling concerns for example try-catch part of the code is moved to the aspect but the error detection part for example 'throw' is not dealt as aspects.

2.3.5 Other Approaches

[Chitchyan et al. 2007] present semantics based composition for requirement engineering to reason about semantic influences and trade-offs among aspects. The approach uses a Requirements Description Language (RDL) based on the semantic information of the natural language. Composition specifications in this approach are based on the requirement specification semantics that also address the reasoning about composition of aspects and trade-offs. The approach is applied on two case studies and supported by the MRAT tool.

In the AOM approach proposed by [Clarke and Walker, 2001][Clarke and Walker, 2002] a design concern, called a Subject, is created for each system requirement and presented as UML model views. Composition relationships are supported by UML metamodels that also describe rules for composition. Merge and override operations are used for the integration. Reconciliation strategies are use to resolve conflicts between property values of corresponding subject elements. Precedence relationships, transformation functions applied to conflicting elements, explicit specification of reconciled elements, and default values may be used for reconciliation.

Similarly, at the implementation level, the work in [Watson et al. 2000] presents an approach with the ability to discover interactions between aspects with the help of Data Flow Analysis (DFA). This approach deals with the error detection mechanism in fault tolerant systems, and the AIDA (Aspect Interference Detection Analysis) tool is used to implement and analyse the interactions for the AspectJ language. AIDA also produces a visual representation of interactions by means of an interaction graph.

2.3.6 Summary

The approaches presented above are aspect oriented development approaches addressing the separation of crosscutting concerns and their weaving. However, many issues still need to be resolved in these approaches for example reasoning about the composition of aspects and crosscutting concerns to avoid conflicts and interactions between them. Many approaches are platform dependent. Moreover, almost all of the approaches only consider exception handling for the fault tolerance and do not focus on other mechanisms such as N-version programming, recovery blocks, etc.

A summary of the work described above is presented in Table 2.2 below. All of these approaches fully support a separation of concern but, as it can be seen from the table; most of approaches partially address the issue of feature interaction analysis.

Approach	AOM	THEME	Motorola WEAVER	AspectJ
System Model	UML, templates, class diagrams, sequence diagram	THEME units, UML Packages, Class diagram	UML classes, state machines, SDL	UML, pointcuts, views, aspects
Composition Mechanism	UML static composition	Theme static composition	Weaver aspects Boolean Operators OR, AND, NOT and <<follows>>	Weaver Weaving process
Feature Interaction Analysis	Partially Supported	Partially Supported	Precedence Constraints	N/A
Platform/paradigm constraint	AOSD	AOSD	C, C++ and Java Aspect-oriented software development	Java, C++
Tool Support/Case Studies	N/A	CMS and Crystal Game	Motorola Weaver, Tau Tool	Telestrada's Complaint Management Subsystem (CMS) Weaver Tool

Table 2.2 Aspect Oriented Modelling and Design Approaches

2.4 Feature Interaction Analysis

The problem of detecting and analysing undesirable interactions is a well-researched area and, in the particular domain of telecommunications systems, research into feature interaction has a long history dating back to the 1990s. Comprehensive surveys exist for this research area, for example those conducted by [Keck and Kuehn 1998] and [Calder et al. 2002].

This section highlights a number of techniques and frameworks that deal with the prediction, detection and resolution of feature interactions as well as the composition of features.

2.4.1 FI Filtering

[Nakamura and Kikuno 2000] present an approach for feature interaction filtering. The approach works at the requirement specification level to identify all possible combinations that have the possibility of feature interaction. The filtering method screens out undesirable feature combinations before the detection process. The method is supported by a requirement notation called Use Case Maps (UCMs) based on scenario paths, for the description of services associated with the UCM Navigator.

In this approach, each feature is expressed as a set of submaps describing scenarios specific to the feature. The rootmap describes the common scenarios with default submaps, into which the feature submaps are plugged. Then the information about feature submap plugging is provided in a stub configuration represented in terms of a matrix, called the configuration matrix. Once each configuration matrix has been characterised, composition of different configurations is carried out by a matrix composition operator based on the semantics of the composition.

System Model: This approach uses the Use Case Maps (UCMs) requirements notation. UCMs provide a system-wide path structure that addresses all possible combinations that have a possibility of feature interactions. FI filtering uses the stub-plug-in concept of UCMs.

Composition Mechanism: The approach uses a matrix composition for composing different configurations. The composition is performed by checking only pre-conditions of the feature submaps. The concepts of fork and join are used to compose the features in stub configurations, where OR fork/join describes an alternative scenario path and AND fork/join describes a concurrent scenario path.

Separation of Concerns: Each feature and configuration is considered separately, hence addressing separation of concerns.

Feature Interaction Analysis: The approach addresses the problem of analysis of feature interactions before detection takes place. FI-prone combinations can be identified but for the FI detection, there is still the need to employ more formal definitions and techniques.

Platform/paradigm constraint: The approach works at the requirement specification level, but is specific for telecommunication systems. The approach is very general and need further work for adoption in other domains.

Tool Support/Case Studies: For designing Use Case Maps, UCM Navigator is used to help to draw syntactically correct UCMs. The FI filtering experiments are carried out on a Basic call model of the Telephony system.

2.4.2 FIN Method (Use Case Driven Analysis)

[Kimbler and Sobirk 1994] present the FIN detection method based on Use Case Driven Analysis of a system at the requirement specification level. There are two models derived from the informal description of services and feature. The first model is use case model that describes the users and services associated with these users in use case. The other model is called service usage model that captures the dynamic dependencies between the services and their features. These dependencies can further work out with the help of service usage graphs. All possible scenarios can be generated from the Service Usage Model.

The feature detection starts from the scenarios generated from the service usage model. This model provides the feature sequence that can be used in the identification of interactions. Similarly, [Felty et al. 2000] also use the FIN tool aided by the model checker COSPAN to check inconsistency between service features. This approach uses formal methods to automatically detect undesirable feature interaction by using linear temporal logic (LTL) at the specification stage.

System Model: This approach uses Use Case Driven Analysis at the requirement specification level to analyse all possible scenarios of the system with the help of Use Case Models and Service Usage Models (SUM). SUMs describe all possible scenarios and automatically analyse them in order to detect interaction-prone feature pairs. These pairs are then manually analysed to determine the occurrence of any undesirable interactions.

Composition Mechanism: The approach does not provide any composition mechanism, and only focuses on feature interaction.

Separation of Concerns: This approach does not explicitly address the issue of separation of concerns.

Feature Interaction Analysis: The FIN detection method fully addresses the problem of analysis and detection of feature interactions with the help of the automated tool and Use Case Driven Requirement specifications.

Platform/paradigm constraint: The approach works at the requirement specification level, but is specific for telecommunication systems and also deals with only functional requirements.

Tool Support/Case Studies: The FIN method is supported by a tool with graphical support for the specification of features and use cases. The tool also supports creation of the service usage graphs and automatic detection of interaction prone feature combinations.

2.4.3 CHISEL

[Turner 2000] describes the CHISEL notation (a graphical language for describing telecommunication services and features) in the context of feature interaction. The CHISEL approach focuses on understanding the feature design first, without the formal notation. CHISEL gives the event sequences that characterise the features with the help of Message Sequence Charts (MSCs), hierarchical textual descriptions, finite state automata, regular expressions and process algebra. Moreover, the extension of CHISEL incorporates the use two formal languages: SDL (Specification Description Language) and LOTOS (Language for Temporal Order Specification). This gives rise to a new name for the extended approach, namely CRESS (CHISEL Representation Employing Systematic Specification).

CRESS uses SDL to specify the features in a system and LOTOS with its synchronous communication provides the interactions between features.

System Model: This approach uses CRESS that has formal denotation given by SDL . However the interpretation of CRESS corresponds closely to an LTS (Labelled Transition System). CHISEL diagrams are translated to Message Sequence Charts, hierarchical textual descriptions, finite state automata, regular expression and process algebra at the

requirement specification level. This helps to address possible combinations that have the possibility of feature interaction.

Composition Mechanism: The approach uses SDL and LOTOS specification for the composition of different features and services. CHISEL diagrams are given a formal representation in SDL and LOTOS. **Feature Interaction Analysis**: LOTOS in CRESS uses synchronous communication that provides the interactions between features. Features are simulated and the detailed exploration of each feature's behaviour is analysed in depth. The verification of desired and undesired behaviour of the feature when composed with a number of other features is carried out with the help of LOLA's TextExpand function.

Platform/paradigm constraint: The approach works at the requirements specification level, but is specific for telecommunication systems. The approach is flexible to adapt for feature interaction analysis in other domains as well.

Tool Support/Case Studies: Features are simulated and analysed for SDL using LOLA (LOTOS Laboratory), along with the detailed exploration of each feature's behaviour in depth. The verification of desired and undesired behaviour of the feature when composed with a number of other features is carried out with the help of LOLA's TextExpand function.

2.4.4 Other Approaches

[Blam et al. 1994] present a formalism for the specification of services and their composition by using first order linear temporal logic. Their methodology is supported by the Z specification language, where logical operators can be used to compose different modules. The aim of this methodology was to reduce deadlocks and composition of services.

[Hay and Atlee 2000] consider each feature as a separate concern and model them as separate labelled-transition systems by defining a conflict-free (CF) composition operator. This approach claims to prevent feature interactions by defining conflict and violation free composition (CVF). [Bredereke 2000] uses formal semantics for detecting feature interactions in the telecommunication domain by using CSP language. The work of [Plath and Ryan 2000] describes the development of a SFI (SMV Feature Integration) tool that changes the traditional SMV [McMillan 1993] code to a compact code with which the model checker can work. This approach describes the detailed semantics for each service and feature construct for the telephony system.

[Kelly et al. 1995] designed two different types of models to investigate feature interactions by using ITU's standard specification language, SDL.

Moreover, there are few online approaches to address feature interaction while features are running. [Marples and Magill 1998], [Reiff 2000] and [Cain 1992] describe the use of a Feature Manager to analyse and detect feature interactions in telecommunication services at run time.

The problem of feature interactions is not limited to the telecommunications domain. Interaction analysis is a well-formed area in the aspect oriented community. The work of [Katz et al. 2008] describes a generic categorisation of aspects which can aid in the analysis of interactions with an automatic detection analysis. Its main application to interference between modules is when advices apply at the same joinpoint, as the categorisation can aid with determining the correct order of application. [Clifton and Leavens 2002] propose a language to see the impact between different modules and provide some interaction analysis in terms of the specifications of the respective advices.

There are also some model-checking approaches to interaction analysis in an aspect orientated context. [Goldman and Katz 2007] present a modular aspect verification technique to analyse the aspects' impact by using linear temporal logic (LTL) formulae. In this approach, the aspect's behaviour can be verified independently from a base system by weaving the aspect in LTL and model checking the result.

A similar approach is presented by [Krishnamurthi et al. 2004]. Model checking is used to determine if the properties of the base system will indeed remain inviolated when the aspect is woven. This approach is useful to detect bad interactions between aspects and base models.

[Kolberg et al. 2003] discuss the issue of compatibility between services in a home environment giving reasons, why and how services interact. This approach further presents a taxonomy of interactions followed by an approach to prevent interactions. The issue of feature interactions is illustrated with a number of scenarios focusing on automatic detection of feature interactions between services in the home. Similarly, [Soares et al. 2012] propose the use of state graphs to predict the feature interactions between off-the-shelf systems. The approach uses pre-deployment simulations to forecast feature interactions before deployment.

2.4.5 Summary

The approaches surveyed above all address feature interaction analysis and detection at the requirement specification and design level, but most of the above approaches deal with the telecommunication domain. Especially, these approaches do not consider the undesirable interactions that arise when fault tolerance have been added.

A summary of these approaches is provided in Table 2.3 below.

Approach	FI Filtering	FIN	CHISEL
Composition Mechanism	Stub-configuration composition	Theme static composition	LOTOS and SDL specifications
Feature Interaction Analysis	Supported	Supported	supported
Platform/paradigm constraint	Telecommunication	AOSD	Telecommunication
Tool Support/Case Studies	UCM Navigator, Telecom	CMS and Crystal Game	SDL, LOTOS, LOLA

Table 2.3: Feature Interaction Analysis

2.5 Comparison and Discussion

An overall comparison of results from this chapter are summarised in Table 2.4. Together, these surveyed studies provide an important insight into the direction of fault tolerance at requirement specification and design phase, aspect oriented design and modelling addressing separation of concerns, feature composition and feature interactions analysis. However, there are also a number of limitations of these approaches, regarding the requirements laid out in chapter 1 for this thesis:

Most of the approaches described in the architecting fault tolerance section provide an explicit description of components, the notion of connectors, configuration, and architectural interfaces and also explicitly deal with exception handling. Most of the surveyed approaches have successfully shown the differentiation between the fault tolerance requirements and functional requirements. But most of them are dealing with a very basic level of exception handling. Other approaches provide formal or model based notations for specifying fault tolerance properties. These approaches also focus on the architecture level and are based on existing languages or notations such as CSP, Petri Nets, B method, etc. Few approaches are supported by automated tools to implement a fault tolerant system, either supporting model-driven development such as DRIP Catalyst, or providing primitives for implementing fault tolerant architectures.

There is still a need to describe the complex fault tolerance mechanisms such as active replication or N version programming with a separation of concerns. Aspect oriented modelling approaches are considered as a good solution for that. It is also believed that the orthogonal view of the fault tolerance with separate concerns can support reusability and easy maintainability of different fault tolerance mechanisms than a large monolithic system.

As can be seen in Table 2.4, the approaches under fault tolerance at architectural level classification partially address the issue of separation of concerns, fault tolerance composition and the feature interaction analysis. Mostly aspect oriented approaches deals with the separation of concerns and the composition of these concerns. However, this is still a big challenge to develop a language for expressing composition strategies and techniques. These approaches generally discuss the aspect weaving but do not addresses specifically fault tolerance concerns at a certain desired level.

Various studies successfully highlight the problem of feature interactions and provide solutions to overcome this problem. These approaches use formal methods, logic properties, and model checking tools, behaviour semantics and software engineering techniques to address the problem of interactions, albeit most commonly in a telecommunications setting. A few of the approaches also discuss the methodology for the composition of features and to some extent compositionality. The problem of compositionality is still a problem especially when fault tolerance features are composed with the system.

Similarly, as shown in Table 2.4, the approaches under feature interaction analysis classification partially address the composition of features. However, these approaches do not address the impact of interactions that arise after composition of fault tolerance concerns to the base system.

In summary the following issues remain unaddressed and hence form the focus of this thesis.

- Design diversity fault tolerance mechanisms: The focus of the thesis is on incorporating design diversity fault tolerance mechanism at the design level based on the requirement specifications. Most of the surveyed approaches deal with exception handling mechanisms, but do not provide any mean to manage different features of design diversity mechanisms. Design diversity fault tolerance mechanisms are comprised of various other mechanisms such as atomic actions,

checkpointing, etc. Hence, it is necessary to provide a broader view of different fault tolerance mechanisms.

- Orthogonal Fault Tolerance Framework at Design Level: Although surveyed approaches deal with fault tolerance at the design level, most are very specific to architectural configurations such as components, connectors, etc. Model driven approaches such as DRIP provides a concept of modularisation and can be a motivational approach that used separation of fault tolerance concerns. However, an orthogonal view of the fault tolerance concerns is missing in most of the surveyed approaches. With the introduction of an orthogonal framework, fault tolerance mechanisms are well separated and can be reused with different components of the base system. The separation of concerns is aimed to reduce software complexity and improve software evolution by keeping track of different fault tolerance mechanisms.
- Composition of Fault Tolerance Mechanisms: The surveyed techniques deal with the composition of crosscutting aspects but not specific to any design diversity fault tolerance mechanism. In particular, when fault tolerance is incorporated in a system, there may be unintended or uncertain interactions which can be resolved by reasoning before composition. The orthogonal view of the fault tolerance mechanism will help in composition of these features, as provides dependency relationships and constraints between features.
- Dealing with interaction: The approaches in the telecommunications domain deal with the problems of feature interaction prevention, detection and resolution. None of the surveyed approaches address the problem of interactions arising when fault tolerance mechanisms are composed with the system. To deal with potential undesirable interactions, it is very important and crucial to know about the features' dependencies. With respect to fault tolerance, the selection of the appropriate mechanism also depends on the clear understanding of components' dependency relationships. Incorporating fault tolerance into a system also brings with it an increased difficulty in system comprehension and increase interaction between different components.

Therefore, although there have been well mature areas such as aspect oriented community and feature interaction in telecommunication, a framework for orthogonal fault tolerance is not yet available that can address the automatic composition of design diversity fault tolerance at the requirement specification and design phase with the automatic composition

and interaction analysis. These areas still provide an inspiration for separation of concern and feature interaction analysis for designing fault tolerance system. In summary, with the inspiriting aspect of each area, the development of such a framework must necessarily integrate the speciality of each area discussed above.

		Fault tolerance expressiveness	Separation of Concern	Feature Interaction Analysis	Fault tolerance composition
Fault Tolerance at Architectural Level	Co-operative Architectural Style	⦶	⦶	⦶	⦶
	Architectural Patterns	⦶	⦶	⦶	⦶
	MAL Specification	⦶	⦶	○	⦶
	AEREAL Framework	⦶	⦶	○	⦶
	iFTElement	⦶	⦶	○	⦶
Model-Driven approach	DRIP Catalyst (CORRECT Project)	⦶	●	○	●
Formal Specification Approach	Event B (DEPLOY Project)	⦶	○	○	○
Aspect Oriented Composition Approach	AOM	⦶	●	⦶	●
	AspectJ	⦶	●	○	●
	Theme	⦶	●	⦶	●
	Motorola WEAVER	⦶	●	⦶	●
Feature Interaction Analysis	FI Filtering	○	⦶	●	⦶
	FIN	○	⦶	●	⦶
	CHISEL	○	⦶	●	⦶

Table 2.4 Overall Comparison Table

Legend: ● Fully Addressed ⦶ Partially Addressed ○ Not Addressed

Chapter 3

Orthogonal Fault Tolerance (OFT) Framework

3.1 Introduction

The orthogonal fault tolerance framework presented in this thesis ensures the separation of concerns between the 'base' system and the fault tolerance mechanisms that are composed with the base system. This orthogonal approach has been inspired by OVM, the Orthogonal Variability Model [Pohl et al. 2006]. The key benefits of OVM are the improvement of decision making, communication and traceability. It also makes the overall development process simple, consistent and unambiguous. This model provides a cross-sectional view of the variability across all software development artefacts [Pohl et al. 2005]. By using a similar orthogonal modelling approach for fault tolerance, it brings similar benefits for fault tolerant software systems.

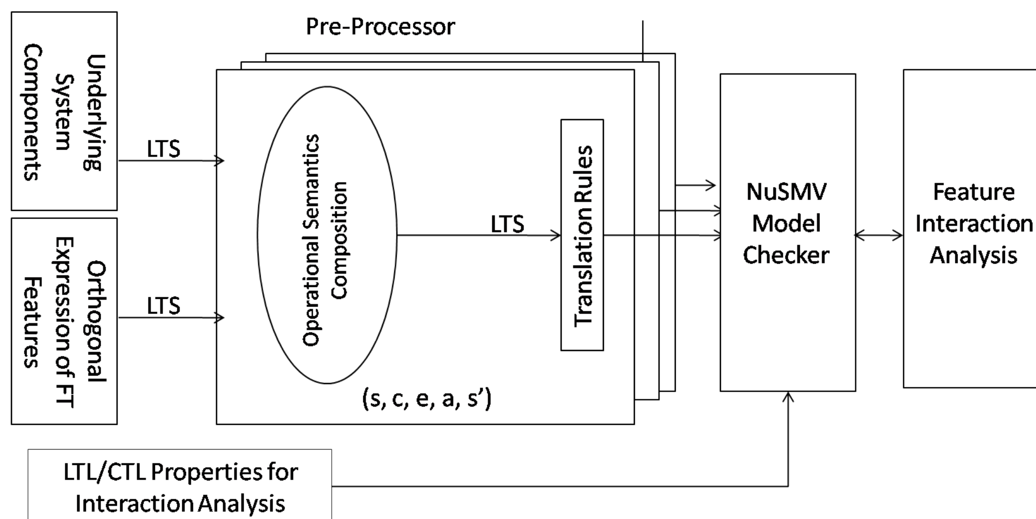
In the proposed orthogonal fault tolerance model, fault tolerance mechanisms are well separated and can be reused with different components of the base system. The separation of concerns aims to reduce software complexity and improve software evolution by keeping track of different fault tolerance mechanisms. The orthogonality in this thesis also establishes the relationships and constraints between features of different design diversity fault tolerance mechanisms based on their dependencies. In addition, analysing the dependencies among features provides essential information for feature selection and integration without impacting on other features of the base system, enabling multiple fault tolerance needs to be managed.

The composition of the orthogonal fault tolerance mechanism with the base system is based on operational semantics that describe the behaviour of the underlying components when composed with the fault tolerance mechanisms. The custom-built pre-processor is based on these composition rules, and is used to automatically compose the system component and the fault tolerance mechanisms. For the pre-processor, features of orthogonal fault tolerance mechanisms are represented as 'generics'; these will be discussed in more detail in

Section 3.3.4. The orthogonal fault tolerance framework is also supported by the model checking tool NuSMV; the automatically generated output of the pre-processor is used as an input for NuSMV.

The very introduction of different fault tolerance mechanisms to the system may cause interactions with other fault tolerance features or with system components. Logic properties written in CTL are used in NuSMV to analyse undesirable interactions.

Figure 3.1 shows the overall context of the proposed approach: a software system in which both the main system component and the fault tolerance features are expressed using Labelled Transition Systems. These two orthogonal parts of the overall system are then composed using a pre-processor underpinned by operational semantics, and this tool automatically generates the input language for the model checker. The models are then verified for correctness, including the absence of inconsistencies and interactions, with the help of logical properties written in CTL in the model checking tool NuSMV.



Legend: s: initial state, c: condition, e: event, a: actions, s': final state

Figure 3.1: Context of the Proposed Method

The rest of this chapter will describe the different elements of the Orthogonal Fault Tolerance Model in greater detail. Section 3.2 contains details of the different features of fault tolerance mechanisms, including subsections on feature dependency analysis and constraints. Section 3.3 provides detail on the composition in the orthogonal fault tolerance framework with labelled transition systems, operational semantics, handling 'generics', composition of components with fault tolerance features and the pre-processor. The feature

interaction analysis with subsections on NuSMV model checking and classification of CTL and logic properties is provided in Section 3.4, followed by a summary in Section 3.5.

3.2 Orthogonal Fault Tolerance Model

As explained in the introductory chapter, the focus of the proposed approach is on software fault tolerance that uses design diversity to build reliable and fault tolerant systems. Design diversity mechanisms are mainly developed to deal with design faults. By using design diversity, a central assumption is that failure is rare in the presence of different software variants. This work is on the design level rather than the implementation and testing level. Table 1 shows the main characteristics of the design diversity software fault tolerance mechanisms that have been considered so far. All of these use multiple versions of the software components that comply with the same specification, but that have different designs and implementations. In addition to multiple versions, four other main features are considered in the Orthogonal Fault Tolerance Model: Judgment Criteria, Execution Scheme, Error Processing Technique and Checkpoints.

Fault Tolerance Mechanism	Features			
	Judgment Criteria	**Execution Scheme**	**Error Processing Technique**	**Checkpoints**
Recovery Block	AT Voter	Sequential Parallel	Backward Forward	Yes No
N-Version Programming	AT Voter	Sequential Parallel	Forward	No
N-self checking Programming	Hybrid	Sequential/Parallel	Forward/ Backward	Yes/No
Distributed Recovery Block	AT	Sequential	Forward	No
Consensus Recovery Block	Hybrid	Sequential/Parallel	Forward/ Backward	Yes/No

TABLE 3.1: Fault tolerance mechanisms features

In Table 3.1, the rows show five common design diversity fault tolerance mechanisms. The different features employed in these mechanisms are shown by the columns of the table.

- The judgment criterion is the adjudicator that helps to decide the correct result among different variants. There are three different adjudicators: Acceptance Test (AT), Voter and Hybrid (combination of both the Acceptance Test and the Voter).
- The variants in different design diversity fault tolerance mechanisms run either sequentially or in parallel. For a Hybrid adjudicator, both execution schemes are used: sequential for the Acceptance Test and parallel for the Voter.

- Regarding the error processing technique, backward error processing normally recovers the system to the previous state whereas forward error recovery brings the system to a new safe state such as with the use of exceptional handling.
- Checkpointing is used to save the system states that will act as recovery points. For the backward error processing, checkpointing is used; the forward error processing technique does not uses checkpointing unless the adjudicator is the hybrid voter.

The details of the dependency constraints and relationships between these different features of fault tolerance mechanisms are presented in the following section.

3.2.1 Feature Dependency Analysis

Before specifying the formalisms for different features of design diversity fault tolerance mechanisms, it is important to define dependency constraints and relationships between them.

The Orthogonal Fault Tolerance Feature Model (see Figure 3.2) is inspired by Brito's feature model for fault tolerance mechanisms [Brito et al. 2009]. The main focus of this model is on the four features of fault tolerance mechanisms (Judgment Criteria, Execution Scheme, Error Processing Technique and Checkpointing) already presented in Table 3.1.

The Orthogonal Fault Tolerance Feature Model (OFTM) captures the various features found in fault tolerance mechanisms reported in literature, as well as the relationships between features. As a result, not only known fault tolerance mechanisms but also bespoke ones can be derived from the OFTM. Relationship constraints such as mutual dependency and mutual exclusivity (depicted by dotted lines in Figure 3.2) are also captured in the OFTM to ensure that only valid combinations of features are included in the fault tolerance mechanism derived from the model.

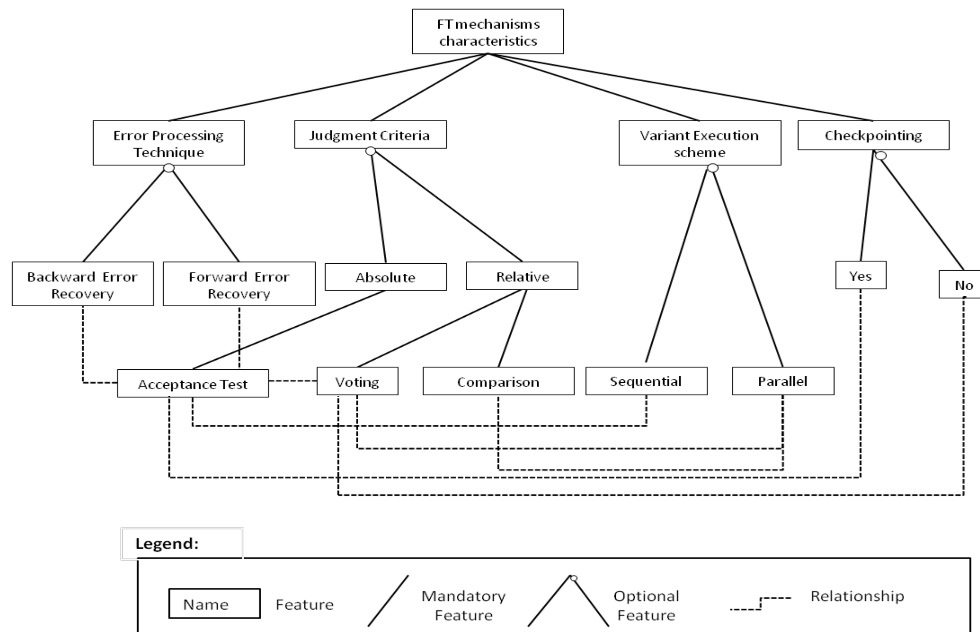


Figure 3.2: Orthogonal Fault Tolerance Feature Model

According to the OFTM, there are four mandatory features of fault tolerance techniques; these relate to the number of versions in use, and also the type of execution scheme, adjudicator and error recovery scheme being used. The mandatory features are as follows:

i. n different Versions of the component, where n can be any number; denoted by VER;

ii. the Execution Scheme, which may be sequential or parallel;
denoted by ES = SEQ ∨ PAR;

iii. an Adjudicator that is used to decide the correctness of the versions' result, which may be Acceptance Test, Voting or Hybrid (a combination of both);
denoted by AD = AT ∨ VOT ∨ HYB;

iv. the Error Recovery scheme, which can be Backward or Forward;
denoted by ER = B ∨ F.

The next section provides the detail of constraints for the valid composition of these fault tolerance features.

3.2.2 Constraints

There are a number of constraints that must be satisfied to ensure that these features are combined in valid ways. The first two constraints involve whether the additional feature of

checkpointing is to be used, while the remaining constraints control the valid combinations of adjudicator and execution schemes.

1. Backward recovery is always accompanied by Checkpointing; $B \Leftrightarrow Ch$;

2. Forward recovery never uses Checkpointing; $F \Leftrightarrow \neg Ch$;

3. Versions running sequentially require an acceptance test as the adjudicator;

   $SEQ \Rightarrow AT$ (i.e. sequential execution cannot use a voter; $SEQ \Rightarrow \neg VOT$);

4. However, use of the acceptance test does not imply sequential execution (since an acceptance test may be part of a hybrid adjudicator for parallel execution);

   $AT \not\Rightarrow SEQ$;

5. Versions running in parallel either use a voter or a hybrid adjudicator;

   $PAR \Leftrightarrow VOT \vee HYB$.

These constraints help in selecting different features from the orthogonal fault tolerance model that is added to the system components.

3.2.3 Typical Design Diversity Mechanisms

Three of the standard fault tolerant mechanisms are Recovery Block, n-Version Programming and n-Self-Checking Programming. The following points show how these can be achieved in relation to the above constraints:

- By combining constraints 1 and 3, it is possible to achieve a *Recovery Block* mechanism. This would consist of n-Versions executing Sequentially with an Acceptance Test and Backward Error Recovery (and hence also Checkpointing);

- Using constraint 5, it is possible to achieve *n-Version Programming* (NVP) that executes in Parallel with Forward Error Recovery and uses either a Voter or a Hybrid adjudicator;

- Also using constraint 5, it is possible to achieve *n-Self-Checking Programming* (NSCP) that executes in Parallel with Forward Error Recovery but that always uses a Hybrid adjudicator.

3.3 Composition in the Orthogonal Fault Tolerance (OFT) Framework

Operational semantics are introduced in order to provide a formal basis to underpin the rules for composition in the OFT framework. Operational semantics are used to show the behaviour of different processes by specifying transitions from a process's state on a particular event. The purpose of developing operational semantics is to give behavioural descriptions of programs and systems, in the form of mathematical formalisms, to support understanding and reasoning about the behaviour of programs and software systems [Prasad 2003]. More specifically, in this thesis, operational semantics are used to provide the rules for the composition of the orthogonal fault tolerance concerns with the underlying system components.

To provide an operational description of a system, Labelled Transition Systems (LTS) are used to denote the execution of events that cause the transitions to occur.

3.3.1 Introduction to LTS (Labelled Transition Systems)

A Labelled Transition System (LTS) is a graph of states and labelled transitions (represented by nodes and edges respectively) which describes the behaviour of a system: in a given state, the possible actions from that state are described, each of which may cause the system to transition into a new state.

A Labelled Transition System is formally represented by a 4 tuple: $< S, E, \delta, s_0>$.

Let S be a non-empty set of states: $S = \{s_0, s_1, \ldots\}$.

Let E be a non-empty set of events: $E = \{e_1, e_2, \ldots\}$.

δ is the transition relation: $\delta: S \times E \times S$ (denoting the Cartesian product of sets).

s_0 is the initial state.

Note that the transition relation $(s, e, s') \in \delta$ is typically written:

$$s \xrightarrow{e} s'$$

that denotes that the LTS moves from state s to state s' on the occurrence of event e.

3.3.2 Operational semantics

Sequential Execution. Assume that an event e is followed sequentially by process P. This sequential operation is denoted by the operator ';' and can be represented as follows:

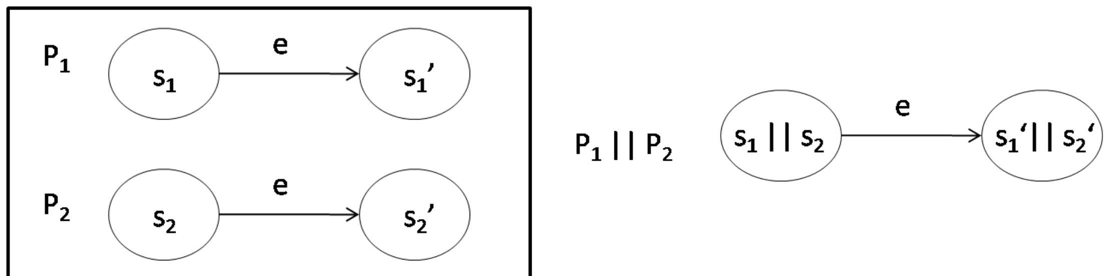
$$\frac{}{e; P \xrightarrow{e} P}$$

Parallel Execution: Full Synchronisation. Assume that two processes, P_1 and P_2, are running in parallel and let s_1 and s_2 denote the states of processes P_1 and P_2 respectively. The full synchronisation behaviour $P_1 \mid\mid P_2$ can be computed by the following inference rules.

If both processes can transition into a new state by performing an event e, then the parallel composition can be described as follows:

$$\frac{s_1 \xrightarrow{e} s_1^{'}, s_2 \xrightarrow{e} s_2^{'}}{s_1 \| s_2 \xrightarrow{e} s_1^{'} \| s_2^{'}}$$

The graphical representation of the above inference rule is as follows:



Other forms of parallel composition can also be defined using operational semantics such as interleaving and composition on selective events [Bolognesi 1987] as follows:

Interleaving. Assume that two processes, P_1 and P_2, are running in parallel and let s_1 and s_2 denote the initial states of processes P_1 and P_2 respectively. The interleaving behaviour $P_1 \mid\mid\mid P_2$ can be computed by the following inference rules.

If process P_1 can transition from s_1 into a new state by performing an event e_1, then the parallel interleaving composition of $P_1 \mid\mid\mid P_2$ can be described as follows:

$$\frac{s_1 \xrightarrow{e_1} s_1^{'}}{s_1 \mid\mid\mid s_2 \xrightarrow{e_1} s_1^{'} \mid\mid\mid s_2}$$

Similarly, if process P_2 can transition from s_2 by performing an event e_2:

$$\frac{s_2 \xrightarrow{e_2} s_2^{'}}{s_1 \mid\mid\mid s_2 \xrightarrow{e_2} s_1 \mid\mid\mid s_2^{'}}$$

Selective composition. Assume that two processes, P_1 and P_2, are running in parallel and should be synchronised on a given set of events, E, where $e_1, e_2, \ldots \in E$. Let s_1 and s_2 denote

the initial states of processes P_1 and P_2 respectively. The selective composition P_1 |[E]| P_2 can be computed by the following inference rules.

If processes P_1 and P_2 can both transition into new states by synchronising on an event e ∈ E, then the selective composition of P_1 |[E]| P_2 can be described as follows:

$$\frac{s_1 \xrightarrow{e} s_1', \; s_2 \xrightarrow{e} s_2'}{s_1|[E]|\, s_2 \xrightarrow{e} s_1' \,|[E]|\, s_2'}$$

However, if one process P_1 can transition into a new state on an event e ∉ E, then the process can progress independently of P_2:

$$\frac{s_1 \xrightarrow{e} s_1'}{s_1|[E]|\, s_2 \xrightarrow{e} s_1' \,|[E]|\, s_2}$$

Note that the interleaving operator is a special case of the selective composition operator |[E]| where E is empty, and that full synchronisation is a special case of the selective composition operator where E contains all possible events.

Choice. If there are two or more alternative behaviours of the process, the choice operator expresses two alternative behaviour descriptions. Suppose s can transition to the state s' on the occurrence of an event e_1. Similarly, s can transition to the state s'' on the occurrence of an event e_2. This is represented by the following interference rules:

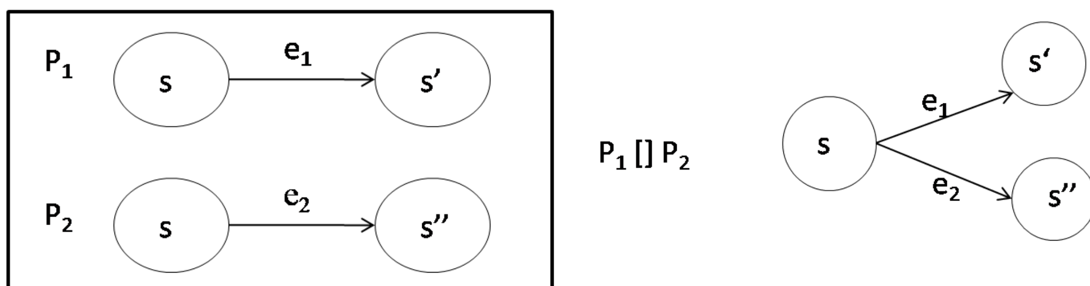
$$\frac{s \xrightarrow{e_1} s'}{e_1; P_1 \,[]\, e_2; P_2 \xrightarrow{e_1} P_1}$$

Similarly for e_2,

$$\frac{s \xrightarrow{e_2} s''}{e_1; P_1 \,[]\, e_2; P_2 \xrightarrow{e_2} P_2}$$

If e_1 and e_2 are identical events, then the choice will be made non-deterministically.

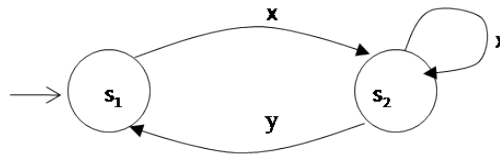
Graphically, the above inference rule can be represented as follows:



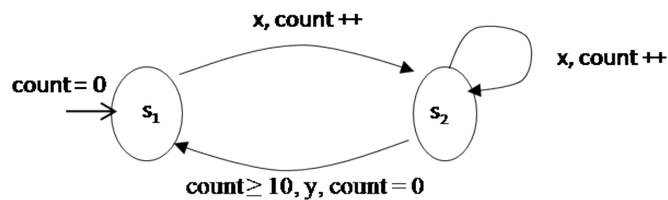
3.3.3 Conditions and Actions over Variables

It will sometimes be necessary to express conditions (guards) over variables to ensure that the transitions only take place under the specified conditions (expressed as predicates over variables). Similarly, actions may be expressed that change the value of variables, e.g. increment/decrement variables.

Consider the following example that represents a sequence of events x followed by an event y.



Suppose that it is necessary to count the number of occurrences of an event x before an event y occurs. This can be expressed by using the integer variable, count. For the purpose of this example, y can only occur if count ≥10, and this event resets the value of count.



As can be seen, this requires the introduction of conditions and actions associated with events. The syntax for these is as follows:

Condition:

Let Var = [a-zA-Z][a-zA-Z0-9_]*

and Op_cond ∈ { <, ≤, ==, >, ≥}

A (Boolean) condition can now be expressed as:

Var Op_cond Var

Action:

As above, let Var = [a-zA-Z][a-zA-Z0-9_]*

let Num = [0-9]*

and Op_act ∈ {+, -, *, /}.

An action can now be expressed as:

Var = Num

| Var Op_act Num

Note that the usual shorthand will be adopted for incrementing/ decrementing the value of a variable, e.g. x = x+1 will be denoted by x++.

It is straightforward to extend the operational semantics to incorporate these conditions and actions over variables. For example, consider the sequential execution rule of 3.3.2:

$$\frac{}{e; P \xrightarrow{e} P}$$

This can be extended with conditions and actions by considering the condition-event-action triple as follows:

$$\frac{}{< c, e, a >; P \xrightarrow{c,e,a} P}$$

This ensures that if the condition c holds, the event e will be performed and will be accompanied by action a.

Similarly, regarding the first composition rule of 3.3.2:

$$\frac{s_1 \xrightarrow{e} s_1^{'}, s_2 \xrightarrow{e} s_2^{'}}{s_1 \| s_2 \xrightarrow{e} s_1^{'} \| s_2^{'}}$$

If one process can now transition into a new state if condition c_1 holds and perform event e and action a_1, while a second process can also transition with event e but on condition c_2 and with action a_2, then the parallel composition can be described as follows:

$$\frac{s_1 \xrightarrow{c_1,e,a_1} s_1^{'}, s_2 \xrightarrow{c_2,e,a_2} s_2^{'}}{s_1 \| s_2 \xrightarrow{c_1\&\&c_2,e,a_1;a_2} s_1^{'} \| s_2^{'}}$$

In the result of this rule, the transition with event e only occurs if the combined constraint $c_1\&\&c_2$ holds; both actions a_1 and a_2 will be performed as a result of the transition. If either c1 or c2 fails to hold, then the transition is blocked.

3.3.4 Handling 'Generics' for Fault Tolerance Components

To be able to handle generics, it is necessary to extend the LTS definition with two new elements: a new state to represent a 'stop' state and a new event τ to represent an internal event. An extended LTS with support for fault tolerance generics is as follows:

A Labelled Transition System is formally represented by a 4 tuple: $< S, E, \delta, s_0>$.

Let S be a non-empty set of states: $S = \{stop, s_0, s_1, \ldots\}$.

Let E be a non-empty set of events: $E = \{\tau, e_1, e_2, \ldots\}$, where τ is an internal event.

δ is the transition relation: $\delta: S \times E \times S$ (denoting the Cartesian product of sets).

s_0 is the initial state.

Note that the transition relation $(s, e, s') \in \delta$ is typically written:

$$s \xrightarrow{e} s'$$

that denotes that the LTS moves from state s to state s' on the occurrence of event e.

The reason for not specifying everything directly as state machines in NuSMV is that a style of specification, termed 'generics' here, is introduced for the specification of fault tolerance components. To illustrate this, a simple example of an Acceptance Test is given below. Further examples of alternative fault tolerance mechanisms will be presented in the context of the case studies in Chapters 4 and 5.

To express the behaviour of an Acceptance Test in a style that is orthogonal to the underlying system, it is necessary to introduce the concept of 'generic' states: states that will later be mapped onto actual states from the underlying system components.

Let the mapping function f: $G \mapsto s$ from the set of generic states G to the set of states S be the function f such that, for every $g \in G$, there exists a unique state $f(g) \in S$.

Let $G = \{g_1, g_2, \ldots\}$ be a non-empty set of generic states such that $G \subseteq S$, where:

$$\exists\, S' \subseteq S \text{ s.t. } \forall s \in S': g \mapsto s$$

i.e. there exists a subset of states from the underlying system onto which each generic state can be mapped.

Let c represent the Boolean condition necessary for the acceptance test to pass,

Let Rp(g) represent the saved recovery point of the generic state g used in the case of the acceptance test failing, $Rp(g) \in S$,

Let τ represent an internal event for transitions on failure to the saved recovery point,

Let n represent the number of software versions.

The necessary transitions relating to an acceptance test with checkpointing (saved recovery points) can now be expressed as follows:

$$g \xrightarrow{c} g'$$

$$g \xrightarrow{!c,\tau,n++} Rp(g)$$

In a given generic state, g, when the acceptance test is passed control moves to a second generic state, g', where both of these states map on to a normal transition in the underlying system, i.e. the fault tolerance component does not affect the underlying system behaviour if the acceptance test is passed. Note that no event needs to be specified in the description of the transition between generic states, since this will be determined by the event offered by the underlying system component (see the first composition rule below).

However, in the case when the acceptance test fails, control moves from the generic state, g, via the internal event,τ, to the saved recovery point. The algorithm for the mapping of generic states on to states from the underlying system will be given in section 3.3.6 below.
Graphically, this can be represented as shown in Figure 3.3:

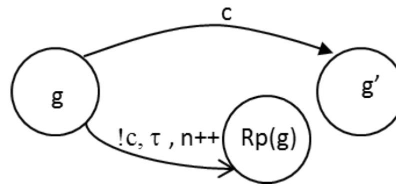


Figure 3.3: A state machine representing an Acceptance Test mechanism and checkpointing

This state machine shows that if the Acceptance Test is passed, then the underlying system component can continue as usual by transitioning from g to a new state g'. However, if the Acceptance Test fails, this means that the current version of the system component will terminate and control will be passed to the next version of the component starting at the saved recovery point.

Using the style of operational semantics presented above, it is possible to give the following two inference rules to allow an underlying system component to be composed with a FT feature such as an Acceptance Test:

$$\frac{s \xrightarrow{e} s', g \xrightarrow{c} g'}{s \,||g \xrightarrow{c,e} s'} \quad \text{(i) acceptance test succeeds}$$

In this first inference rule, if the Acceptance Test is passed (c is true), then the state of the underlying system component progresses as usual ($s \overset{c,e}{\rightarrow} s'$), and the generic states g and g' are mapped to the concrete states (s and s'), and then disappear as shown in above interference rule result.

The purpose of introducing fault tolerance mechanisms is to eliminate the system transitioning to a stop state or inconsistent state on failures and instead bring the system to a safe state.

In the second inference rule below, if the Acceptance Test fails (c is false), then the state of the underlying system component moves (via an internal action) to the saved recovery point represented by RP(s). With the help of 'generics', g is mapped on to a concrete state s: $g \mapsto s$. Similarly for the recovery point, the generic Rp(g) is mapped on to Rp(s).

$$\frac{g \xrightarrow{!c,\tau,n++} \mathrm{Rp}(g)}{s\,||g \xrightarrow{!c,\tau,n++} \mathrm{Rp}(s)} \quad \text{(ii) acceptance test fails}$$

However, at this low level of abstraction there is insufficient information to fully represent the *different versions* of components that are required for fault tolerance, so it is necessary to introduce a higher level of abstraction, through processes (instead of states).

Let process P_i represents the behaviour of component version n when starting at the initial state, and let $P_i[s_i]$ represent the behaviour of component version i when starting at state s_i. P_i' represents the new behaviour of this component once an event has been completed.

Similarly, let process F represents the initial behaviour of the fault tolerance component, and F' represent the behaviour once the Acceptance Test has passed. In the case of the Acceptance test failing, RP(F) denotes the recovery point of the fault tolerant component from its initial state g. In this case, the next version of the process P_i needs to be executed, i.e. P_{i+1}.

It is now possible to rewrite both of the above inference rules to include the required level of information regarding different versions of components:

$$\frac{P_i \overset{e}{\rightarrow} P_i'\,,\, F \overset{c}{\rightarrow} F'}{P_i||F \overset{c,e}{\rightarrow} P_i'} \quad \text{(iii) acceptance test succeeds}$$

With this first inference rule, only the level of abstraction has changed. If the Acceptance Test is passed (c is true), then process P_i (representing the underlying system behaviour)

progresses as usual by performing event e and transitioning to process P_i'. As above, the transition of the generic process F to F' does not require an event, since the event is determined by the underlying system component.

However, consider the above case where the Acceptance Test fails. If states are simply replaced by processes, the second inference rule would become the following:

$$\frac{F \xrightarrow{!c,\tau,n++} Rp(F)}{P_i \parallel F \xrightarrow{!c,\tau,n++} Rp(P_i)} \quad \text{(iv) acceptance test fails}$$

However, this makes no reference to the next version of the process, P_{i+1}, needing to be invoked from the saved recovery point. Hence, the following inference rule is introduced:

$$\frac{F \xrightarrow{!c,\,\tau\,,n++} Rp(F),\ \exists\, P_{i+1}}{P_i \parallel F \xrightarrow{!c,\,\tau\,,n++} P_{i+1}[Rp(P_i)]} \quad \text{(v) acceptance test fails with versions}$$

This new inference rule takes into account the sequential execution (≫) of different versions of the component.

If process P_i has transitioned into a recovery point Rp (indicative of a failed acceptance test), then process P_{i+1} must be instantiated sequentially (assuming variation i+1 of the process exists). This new process P_{i+1} will be started at the saved recovery point represented by $Rp(P_i)$

In the case that P_{i+1} does not exist, the composition will be stopped.

$$\frac{F \xrightarrow{!c,\,\tau\,,n++} Rp(F),\ \nexists P_{i+1}}{P_i \parallel F \xrightarrow{\tau} stop} \quad \text{(vi) no versions left}$$

Examples of Voting and Hybrid fault tolerance mechanisms will be presented in the context of the case studies in chapters 4 and 5, but it is worth noting here that these can be expressed in a similar way with a variable 'r' that represents the voting result and both 'c' and 'r' representing the hybrid voter as follows:

$$\frac{s \xrightarrow{e} s',\ g \xrightarrow{r} g'}{s \parallel g \xrightarrow{r,e} s'}$$

With the Voter, the result 'r' replaces the AT condition 'c'.

$$\frac{s \xrightarrow{e} s',\ g \xrightarrow{c,r} g'}{s \parallel g \xrightarrow{c\&\&r,e} s'}$$

With a Hybrid Voter, both 'r' and 'c' are used.

It can be noted that with the introduction of the orthogonal fault tolerance mechanism, in the case of failure, the composition of the underlying system component with the fault tolerant feature ensures that the system stays in a safe state (represented by the saved recovery point). An example showing the implementation of this composition through the custom built pre-processor is now presented below.

3.3.5 Component Composition with Fault Tolerance Conditions

As stated in Section 3.3.3 above, components can be composed in parallel with conditions taken into account; if both conditions hold then composition proceeds as stated in the composition rule given above:

$$\frac{s_1 \xrightarrow{c_1,e,a_1} s_1^{'} , s_2 \xrightarrow{c_2,e,a_2} s_2^{'}}{s_1 \| s_2 \xrightarrow{c_1\&\&c_2,e,a_1;a_2} s_1^{'} \| s_2^{'}}$$

Note that this assumes the sequential execution rule for the condition-event-action presented in Section 3.3.3 above:

$$\frac{}{< c, e, a >; P \xrightarrow{c,e,a} \mathrm{P}}$$

However, if one or more fault tolerance conditions fail, then the synchronised transition on event e is blocked, and a component's recovery points must be taken into account. The inference rules for this kind of behaviour are as follows:

$$\frac{s_1 \xrightarrow{!c_1,\tau} \mathrm{Rp}(s_1)}{s_1 \| s_2 \xrightarrow{!c_1,\tau} \mathrm{Rp}(s_1) \| s_2}$$

And similarly,

$$\frac{s_2 \xrightarrow{!c_2,\tau} \mathrm{Rp}(s_2)}{s_1 \| s_2 \xrightarrow{!c_2,\tau} s_1 \| \mathrm{Rp}(s_2)}$$

3.3.6 Pre-Processor Tool (Lex & Yacc)

The Orthogonal Fault Tolerance framework provides the separation of fault tolerance concerns and the base system. The features of different fault tolerance mechanisms are handled independently and composed with the base system component with the help of operational semantics presented above (3.3.2 and 3.3.4).

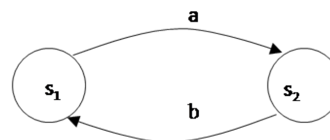
This composition has been implemented as a Pre-processor, to be executed before the use of the model checking tool NuSMV, and has been written in Lex and Yacc. The role of this Pre-processor is to compose fault tolerance generics with the base system components and produce a file representing the composed system that is suitable for input into NuSMV. The reason for selecting Lex and Yacc for this implementation is its rapid application prototyping, easy modification and simple maintenance of the program.

During the first phase, the pre-processor reads the specification input file (describing states, events, transitions and initial states for the system components and fault tolerance features), and uses regular expressions to scan and match strings and convert them to tokens. This stage generates a file lex.yy.c, which contains C code for the Lexer. In the second phase, Yacc uses a context free grammar to generate the constructs required for the NuSMV input file. To parse an expression, shift-reduce parsing is used. This phase generates the files y.tab.h and y.tab.c. Both of these c files are compiled and produce the required output file (in NuSMV format).

The textual and graphical representation of a simple example for a Recovery Block mechanism is as follows. Note that a Recovery Block mechanism involves n-Versions executing sequentially with an acceptance test and backward error recovery (and hence also recovery checkpoints) – as mentioned in the description of a Recovery Block mechanism in the first bullet point of Section 3.2.3.

Spec: simple.txt

```
Define Comp;
states = {s1, s2};
events = {a, b};
start  = s1, a; // initial state &event for NuSMV
transitions = {
            s1, _, a, _, s2;
            s2, _, b, _, s1};
```

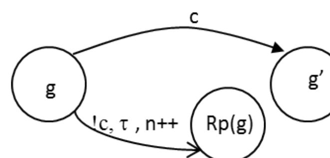


The specification file for the acceptance test is as follows:

Spec: at.txt

```
g, !c, _, n++, Rp(g);};
```

```
Define FT;
states = {g, g'};
condition= {c};
n = 1;  // number of software version
max_n = 3; // max number of versions
start  = g, c;
transitions = {
            g, c, _, _, g';
```



Note: The integer variable, n, is used to represent software versions and can be incremented on the failure of one version. The variable g represents the generic states onto which any state from the underlying component can be mapped on. For the purposes of this simple example, it is also assumed that the saved recovery point is set to be the state from which the failed transition started, but executed on the next sequential version of software.

In Yacc, an algorithm (represented by the pseudocode below) starts by finding all transitions involving generic states from the fault tolerance feature. The algorithm then searches for all possible matching transitions from the underlying system component, and maps the generic transitions onto these component transitions. The system transitions are then augmented with the extra information from the fault tolerance feature; this can take the form of adding conditions and actions, and/ or changing the destination state. The overall result of this for the acceptance test is the inclusion of the additional guard condition ‘c’. In the case of the failure of this condition, the software version being executed will also be incremented and recovery will be from the state represented by the saved recovery point.

This algorithm is based on the operational semantics provided in section 3.3.2 with the choice of implementing parallel composition or sequential composition and also considering conditions and actions for the fault tolerance acceptance test condition and incrementing variants respectively.

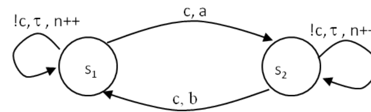
```
executeVer:
if (n<= max_n)
for (startGeneric in FT.states)                    // loop through all FT states
// search for matching FT transitions
        genericTransitions = searchTransitions(FT, startGeneric)
        if (genericTransitions != null)
        // at least one matching generic transition exists
        for (genericTransition in genericTransitions)
                for (startComp in Comp.states)                    // loop through all Comp states
                // search for matching Comp transitions
                compTransitions = searchTransitions(Comp, startComp)
                if(compTransitions != null)
                        // at least one matching Comp transition exists
                        // ... so the generic transition(s) can map onto it/them
                        for (compTransition in compTransitions)
                        // need to add c and a from the generic transition to the comp transition
                                compTransition.condition = compTransition.condition
                                                        &&genericTransition.condition
                                compTransition.action = compTransition.action
                                                        : genericTransition.action
                                if (isNotRecoveryPoint(genericTransition.endState))
                                        // leave start/end states from Comp as they stand
                                else if (isRecoveryPoint(genericTransition.endState))
                                        // end state needs to change to recovery point
                                        compTransition.endState
                                                =recoveryPoint(compTransition.startState)
                                        thisVersion.stop;
                                                executeVer(n+1, compTransition.endState)
```

Pseudocode1: Handling generics for a Recovery Block mechanism

Execution of this algorithm results in the composition of simple.txt and at.txt as illustrated in the summary below:

$$g \xrightarrow{c} g', s_1 \xrightarrow{a} s_2 \Rightarrow s_1 \xrightarrow{c,a} s_2$$

$$g \xrightarrow{c} g', s_2 \xrightarrow{b} s_1 \Rightarrow s_2 \xrightarrow{c,b} s_1$$



$$g \xrightarrow{!c,\tau,n++} \mathrm{Rp}(g), s_1 \xrightarrow{a} s_2 \Rightarrow s_1 \xrightarrow{!c,\tau} \mathrm{nextVer}: s_1$$

$$g \xrightarrow{!c,\tau,n++} \mathrm{Rp}(g), s_2 \xrightarrow{b} s_1 \Rightarrow s_2 \xrightarrow{!c,\tau} \mathrm{nextVer}: s_2$$

The above algorithm illustrates the composition of an orthogonal fault tolerance acceptance test with an underlying system component. The context of this acceptance test mechanism is one of the sequential execution of the different software versions, and the use of saved recovery points (checkpoints). When the Acceptance Test condition is passed, the system transitions to a new state s_2, but when the Acceptance Test condition is not true, the system recovers to the previous state that is the recovery point, but executed on the next version of software. The output of the pre-processor is produced in a form that is suitable for input into the model-checking tool NuSMV. The input format and the output produced (composed

FSMs with fault tolerance and model checking in NuSMV screenshots) are shown with the help of case studies and are presented in Appendices A & B.

3.3.7 Revisiting other Fault Tolerance Features handled by 'Generics'

A similar algorithm is used in the cases of n-version programming or n-self-checking programming, where a parallel execution scheme is used along with either Voting or a Hybrid adjudicator (as described in Section 3.2.3).

More generally, in this section the different fault tolerance features presented in Table 1.3 are revisited to show how they are handled by 'generics'.

- **Judgment Criteria**: The acceptance test, voter and hybrid voter can be handled by 'generics' as already presented in detail in Section 3.3.4.

$$g \xrightarrow{c} g' \Rightarrow s_1 \xrightarrow{c,e} s_2 \quad \text{acceptance test}$$

The generic FT component imposes a condition on a transition between states of the underlying system component. For example, a motion_detected transition $s_1 \xrightarrow{\text{motion_detected}} s_2$ would become $s_1 \xrightarrow{\text{AT-condition, motion_detected}} s_2$ when composed with the FT component that requires an Acceptance test to be passed.

Similarly, in case of voter, the generic FT component imposes the voting result on a transition between states of the underlying component.

$$g \xrightarrow{r} g' \Rightarrow s_1 \xrightarrow{r,e} s_2 \quad \text{voter}$$

For example, a motion_detected transition (when composed with the generic voting component) would become $s_1 \xrightarrow{\text{Voting-result, motion_detected}} s_2$

However, in case of Hybrid voter, both AT-condition as well as Voting-result are imposed on a transition between states of the underlying component.

$$g \xrightarrow{c,r} g' \Rightarrow s_1 \xrightarrow{c\&\&r,e} s_2 \quad \text{hybrid voter}$$

For example, a motion_detected transition (when composed with a generic hybrid component) would become $s_1 \xrightarrow{\text{AT-condition \&\& Voting-result, motion_detected}} s_2$

- **Execution Scheme:** With the introduction of an integer variable, n, execution schemes can be specified such as a sequential execution scheme, where the increment of variable n specifies that the next sequential version is executed:

$$g \xrightarrow{!c,\tau,n++} \text{Rp}(g), s_1 \xrightarrow{e} s_2 \Rightarrow s_1 \xrightarrow{!c,\tau} \text{nextVer}: s_1$$

Similarly, in a parallel execution scheme, all n versions can be executed simultaneously and generate a result ‘r’ based on a majority voting algorithm. The forward error recovery scheme in parallel execution scheme can be specified as follows, where exc(s) represents a new forward safe state represented by exception handling:

$$\mathrm{g} \xrightarrow{!\mathrm{r},\tau} \mathrm{exc(g)}, s_1 \xrightarrow{\mathrm{e}} s_2 \Rightarrow s_1 \xrightarrow{!\mathrm{r},\tau} \mathrm{exc}(\mathrm{s}_1)$$

- **Error Processing Technique:** As presented in Section 3.3.3, there are two error processing techniques: backward error recovery and forward error recovery. In the case of backward error recovery, the execution transitions to the state represented by the recovery point (typically the previous state), but executed (as above) on the next sequential version:

$$\mathrm{g} \xrightarrow{!\mathrm{c},\tau,\mathrm{n}++} \mathrm{Rp(g)}\,, s_1 \xrightarrow{\mathrm{e}} s_2 \Rightarrow s_1 \xrightarrow{!\mathrm{c},\tau} \mathrm{nextVer:s}_1$$

  In the case of forward error recovery, if the voting does not produce a majority result then the system transitions to a new safe state, for example by using exception handling. This new safe state needs to be introduced into the specification of the fault tolerance feature. Suppose this new safe state is represented by exc(g) that will be mapped on to a new state from the underlying system component:

$$\mathrm{g} \xrightarrow{!\mathrm{r},\tau} \mathrm{exc(g)}, s_1 \xrightarrow{\mathrm{e}} s_2 \Rightarrow s_1 \xrightarrow{!\mathrm{r},\tau} \mathrm{exc}(\mathrm{s}_1)$$

- **Checkpointing:** As explained in the constraints Section of 3.2.2, checkpointing is always dealt with in the context of backward error recovery and recovery points and can therefore be presented as above, as if the acceptance test condition fails:

$$\mathrm{g} \xrightarrow{!\mathrm{c},\tau,\mathrm{n}++} \mathrm{Rp(g)}\,, s_1 \xrightarrow{\mathrm{e}} s_2 \Rightarrow s_1 \xrightarrow{!\mathrm{c},\tau} \mathrm{nextVer:s}_1$$

If checkpointing is taken into account for the parallel execution scheme, it can be represented as follows:

$$\mathrm{g} \xrightarrow{!\mathrm{r},\tau} \mathrm{Rp(g)}\,, s_1 \xrightarrow{\mathrm{e}} s_2 \Rightarrow s_1 \xrightarrow{!\mathrm{r},\tau} \mathrm{Rp}(s_1)$$

In this case, the control will not pass to the next version as all versions are executed in parallel, but in case of failure the previous safe state will be retrieved.

3.4 Feature Interaction Analysis

Similar to introducing fault tolerance at the requirement specification and design level, early analysis and detection of feature interactions can help prevent costly and time consuming problems at the later stages of software implementation. It is therefore essential to address and reason about these interactions as early as possible, such as at the requirement specification and at the design level.

[Calder et al. 2000] have presented a critical review on feature interactions concentrating on three different research trends; software engineering, formal methods and online techniques. Background research in this area has been covered in Chapter 2.

In the OFT framework, fault tolerance features are specified separately, and later composed with the system's components to offer reliability. However, sometimes the orthogonal nature of this approach may cause undesirable interactions. The proposed approach for handling such feature interaction analysis has three parts, as described in Fig 3.4:

1. Classification of Categories of Interactions: it is essential to have a sound classification towards a general scheme for detecting and analysing undesirable feature interactions.
2. Specification and ability to Reason about Interactions: reasoning about these classifications is achieved in terms of specifying desirable/ undesirable properties in CTL.
3. Tool Support: the model-checking tool NuSMV is used to support the analysis of feature interactions.

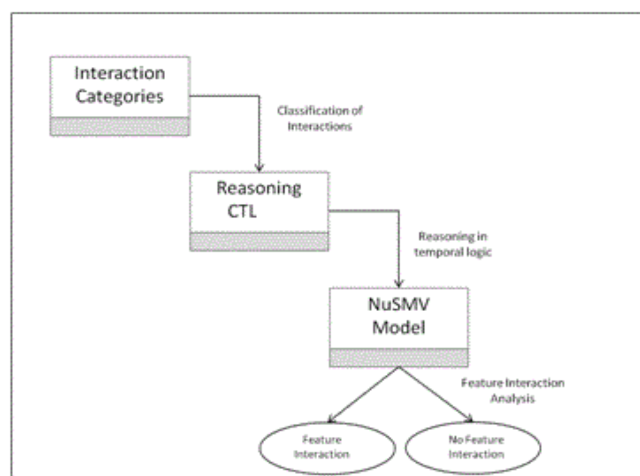


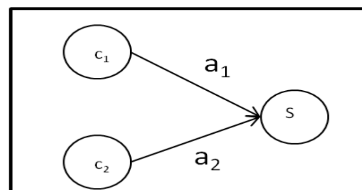
Figure 3.4: Workflow of Proposed Feature Interaction Analysis Approach

3.4.1 Categories of Feature Interaction

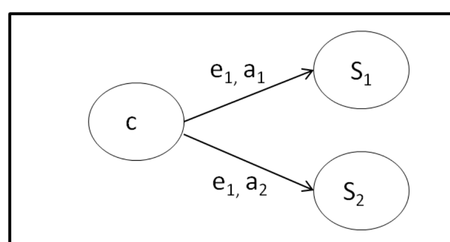
In traditional feature interaction research such as [Kolberg et al. 2003] and [Soares et al. 2012], feature interactions are helpfully classified according to categories such as:

(i) **Multiple Action Interaction:** These types of interaction arise when two different components are controlling the same service (or device), but simultaneously invoke it with different actions. For example, one component may want to turn a device off whilst the other component wants to turn it on.

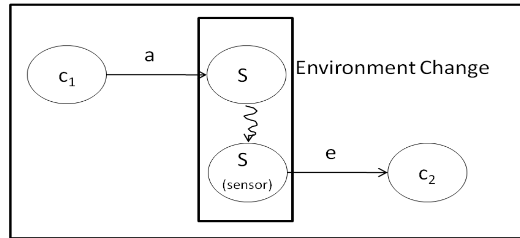
The figure below shows this type of interaction, where C represents the components and S represents the service.



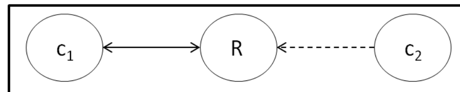
(ii) **Shared Triggered Interaction:** This type of interaction arises when a component sends the same event to two different services (or devices) that then perform conflicting actions. An example might be when a motion detected event is sent to two different services that control different elements of a home automation system – one service might try to turn an automatic light service on, whilst the other might turn an intruder alarm service on.



(iii) **Sequential Action Interaction:** In this type of interaction, the first component changes some element of the environment, which in turn may trigger a second component. For example, a component controlling the temperature in a room may lead to a window being opened, but this movement within the environment may trigger an intruder alarm.



(iv) **Resource Contention:** Resource contention interactions occur when the number of requested resources is greater than the number of available resources. In the case where the number of allocated resources is equal to the number of available resources, an additional request to the resources will result in a resource contention interaction. In the figure below, R represents a single resource and C_1 and C_2 represent the components trying to interact with this resource.



(v) **Assertion Invalidation:** The expected and intended behaviour of the system can be represented through assertion properties, which can later be validated through model-checking. This type of interaction is detected when a component or feature intention is violated in the system. In other words, an assertion invalidation interaction occurs when an assertion made for the system's behaviour is invalidated or false. An example is:

AG ($p \rightarrow q$) which means that whenever property p has been raised, property q must also have been raised.

If this assertion p is true and q is false, it means the assertion is violated and the source of this violation needs to be tracked down. It is possible (though not necessary) that the source of the interaction is one of the above categories, such as shared trigger interaction or sequential action interaction.

3.4.2 Specification of properties in CTL

Logic properties are formulated in CTL (computation tree logic), allowing the analysis of interactions, deadlocks and other correctness properties such as liveliness and safety.

Logic properties are formulated based on the classification of interaction presented in Section 4.1. Typically, these can be specified as safety properties of the system that indicate that an undesirable property would never occur.

CTL is used to represent the properties with standard logical operators (∧, ∨, ¬, ⇒, ⇔, =, <, >, etc.) and temporal operators. A summary of the temporal operators follows:

Always: AG p, EG p

The formula AG p describes that for all the possible paths of the system, property p is always (or globally) true.

In contrast, the formula EG p requires that there is some path of the system along which property p is always true.

Eventually: AF p, EF p

In the logic formula AF p, for all the paths of the system, eventually (at some point in the future) property p must hold. That is, all the possible evolutions of the system will eventually reach a state satisfying property p.

Similarly, logic formula EF p requires that there exists some path of the system that eventually satisfies property p.

Until: A [p U q], E [p U q]

According to the formula A [p U q], in all paths of the system property p must be true until a state is reached that satisfies property q.

With E [p U q], there must exist a path through the system in which property p is true until a state is reached that satisfies property q.

Next: AX p, EXp

The formula AX p requires that for all possible paths through the system, property p is true in all next states reachable from the current state.

Similarly EX p requiring that in at least one path through the system, property p is true in a next state reachable from the current state.

3.4.3 Reasoning about Feature Interactions through CTL and Model Checking

The proposed method of reasoning about the above classifications of Feature Interactions is based on the specification of properties in CTL, and then using model checking with the support of the tool NuSMV. This aims to detect each type of feature interaction defined in the classification above.

Previous model checking based feature interaction detection methods have, for example, used the SPIN model checker (Promela) for web services [Zhang et al. 2007] and LOTOS and SDL for Telecommunications [Blom et al. 1992]. However, for the OFT framework, the NuSMV model checker is used.

CTL formulae can be used to model-check for the above mentioned styles of feature interaction as represented below, where 'p' represents the property where the first component controls the service, and 'q' represents the property where the second component controls the service.

For **Multiple Action Interactions**, the temporal operator AG can be used, for example:

AG ! (p & q)

According to this property, it should never be the case that the two components C_1 and C_2 are controlling the same service at the same time by requiring conflicting properties (actions) to be true.

For example, ensuring that one component does not turn a device off whilst the other component turns it on: AG ! (on & off).

For **Shared Trigger Interactions**, one component controls a service through one event, and a second service based on the same event. A formula to ensure this does not occur can be represented as follows:

AG! (p -> AF(q&r))

For example, a motion detection event might cause one service to try and turn an automatic light service on, whilst the other might turn an intruder alarm service on. This could be represented as:

AG! (motionDetected ->AF(automaticLights_on & intruderAlarm_on))

For **Sequential Action Interaction**, if the first component changes the environment, this change triggers a second component. This style of interaction can be represented as follows:

AG((p&q) -> AFr)

For example, a Climate Control component opens and closes windows according to the temperature, which might cause a sensor to detect motion due to environment change; this may trigger other components to take action according to this motion detection such as initiating a lighting by presence feature and turning the intruder alarm on. This type of interaction can be represented as:

AG((windowOpen & motionDetected) ->AF(lightingByPresence & intruderAlarm_on))

For **Resource contention,** a temporal logic AG can be used as follows:

AG ! (p & q)

According to this property, it will never be the case where two components can hold the resource at the same state. For example, in the case of a shared printer between two components, only one can be the state of printing while other needs to be in waiting state until the printer is free. This example can be represented as follows:

AG ! (c_1. state = printing & c_2.state = printing)

For **Assertion Invalidation**, different combinations of temporal and Boolean logics can be used depending on the assertions made according to the system's intended and expected behaviours. For example in the above case of a shared printer, suppose that whenever a component has requested the printer, it will eventually obtain it. This assertion can be represented with temporal logic as follows:

AG ((c_1. state = requesting & waiting) ->AF (c_1.state = printing))

In this example, if the logic formula is false, that means assertion is violated. Furthermore, the reason of this assertion violation will help to find out the resolution of this assertion raised.

Further illustration of using CTL formulae with the worked example of home automation and the safety property in Therac-25 case study will be presented in Chapter 4 and in Chapter 5 respectively.

3.5 Summary

An Orthogonal Fault Tolerance (OFT) framework has been outlined in this chapter. With the help of this framework, fault tolerance features can be composed with the components of the underlying system at the early stages of software development. Moreover, fault tolerance features are considered as separate concerns and their composition is underpinned by operational semantics. Using a logic-based feature interaction analysis method, CTL logic properties are specified to analyse and detect feature interactions with the support of the NuSMV model checking tool. A summary of the proposed approach with feature interaction analysis is presented in Fig 3.5 below.

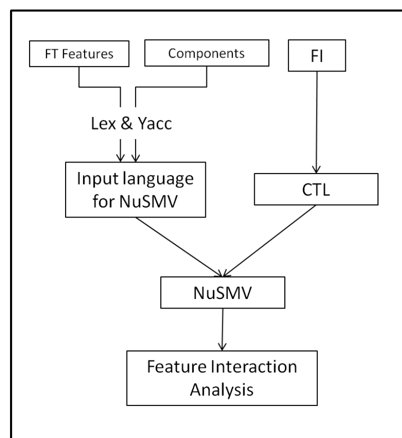


Figure 3.5: Model Checking Based Feature Interaction Analysis

Chapter 4

Illustrative Case Study: A Home Automation System

4.1 Introduction

The illustrative example that will be used to highlight the key elements of the proposed approach is that of a simple Home Automation scenario, where a house is fully equipped with a set of electrical sensors and actuators. Figure 1 below is a Feature Diagram for a smart home, which provides security and illumination services.

Initially this example will be presented with no fault tolerance mechanisms incorporated. These will be considered once the basic example and relevant formalisms have been introduced.

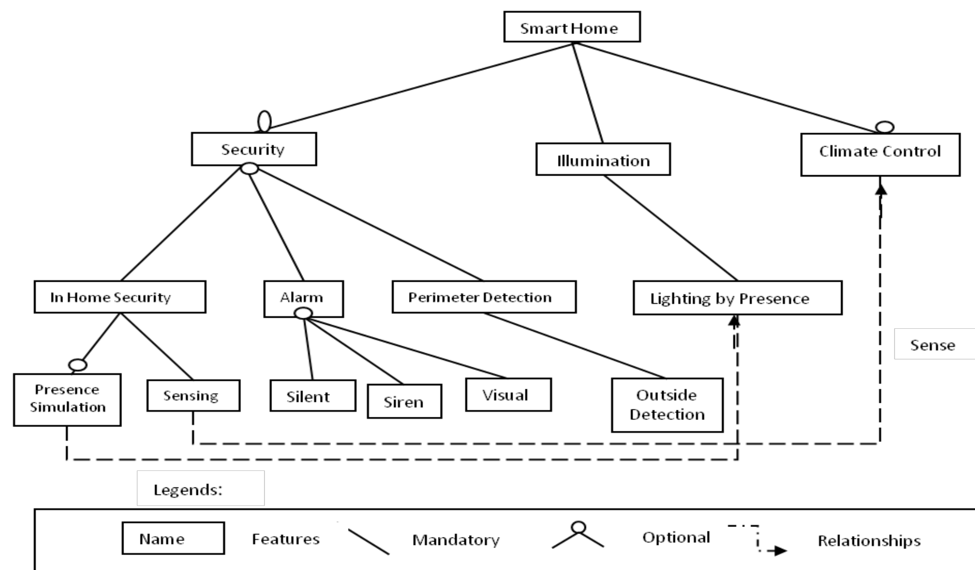


Figure 4.1: A Feature Diagram for a Smart Home

Grey boxes and black solid circles are used to show the mandatory features while white boxes are optional features.

As an example, assume that the Illumination feature *Lighting by Presence* is on. The purpose of this feature is to automatically turn the lights on. It detects motion by using a motion sensor. This links to the In Home Security Feature, *Presence simulation,* which simulates presence in the home by automatically turning lights on. Finite state machines will be used to show the behaviour of different features of the system.

Figure 4.2a shows the Light Controller (LC) component of the Home Automation System.

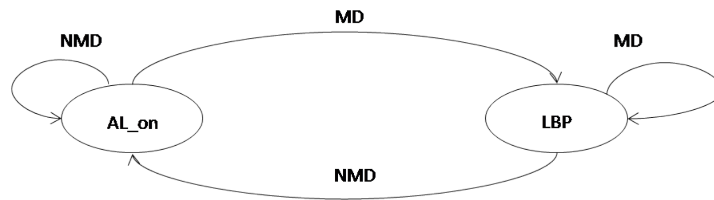


Figure 4.2a: Light Controller (LC) State Machine

Key: MD: Motion Detection, NMD: No Motion Detection, LBP: Lighting by Presence, AL_on: Automated Lights On,

This state machine shows that, from the initial Automatic Lights On state, if no motion is detected then the system stays in the same state, but once motion is detected then the system transitions into a Lighting By Presence state. From this new state, if motion is detected then the system stays in this state, but if no motion is detected then the system transitions back into the Automatic Lights On state.

Now consider a scenario where a new feature, Presence Simulation (for security) is required. This feature simulates that the home is occupied by turning lights on and off while the occupants are away. The Home Status Controller (HSC) component identifies the status of the home: either empty or occupied. This status is determined by using a motion detector sensor.

Figure 4.2b shows the Home Status Controller (HSC) component of the Home Automation System.

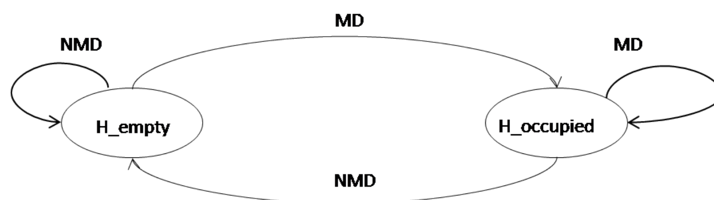


Figure 4.2b: Home Status Controller (HSC) State Machine

Key: MD: Motion Detection, NMD: No Motion Detection, H_empty: Home Empty, H_occupied: Home Occupied

In order to model the *Presence Simulation* feature, if the home is empty, then the Automated Lights feature will be enabled, otherwise the Lighting by Presence feature will be utilised. Both of the components (Light Controller (LC) and Home Status Controller (HSC)) use a motion detection sensor to perform the required task.

Figure 4.3 below shows the initiative (and manual) composition of these two state machines. These components are synchronising on the common events: motion detected and no motion detected, i.e. using the motion detection sensor to control both the home status and the home illumination.

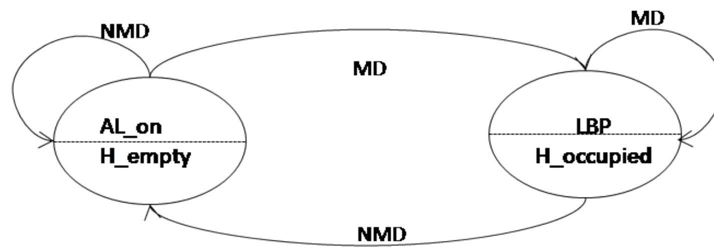


Figure 4.3: Composed State Machine for LC and HSC

Key: MD: Motion Detection, NMD: No Motion Detection, LBP: Lighting by Presence, AL_on: Automated Lights On, H_empty: Home Empty, H_occupied: Home Occupied

4.2 Formalisms /Operational Semantics

Operational semantics give a precise description of the behaviour of a program or a system, and are used to give behavioural descriptions in a mathematical form that supports understanding and reasoning about the system's behaviour. (see Section 3.3.1)
This section will show the application of Labelled Transition Systems (LTS) to the Home Automation.

4.2.1 Applying Labelled Transition Systems to the Home Automation System

A Labelled Transition System (LTS) for the Light Controller (LC) is a 4 tuple $<S_{LC}, E_{LC}, \delta_{LC}, s_{LC0}>$.

Let S_{LC} be the set of states $S_{LC} = \{AL_on, LBP\}$, where the initial state $s_{LC0} = AL_on$.

Let E_{LC} be the set of events $E_{LC} = \{MD, NMD\}$.

Let δ_{LC} be the transition relation; the set of transition relations can be written as:

$T_{LC(MD)} = \{AL_on \xrightarrow{MD} LBP, LBP \xrightarrow{MD} LBP\}$ and

$T_{LC(NMD)} = \{LBP \xrightarrow{NMD} AL_on,\ \ AL_on \xrightarrow{NMD} AL_on\}$

Similarly, a Labelled Transition System for the Home Status Controller (HSC) is a 4 tuple <S_{HSC}, E_{HSC}, δ_{HSC}, s_{HSC0}>.

Let S_{HSC} be the set of states S_{HSC} = {H_empty, H_occupied}, where s_{HSC0} = H_empty.

Let E_{HSC} be the set of events E_{HSC} = {MD, NMD}.

Let δ_{HSC} be the transition relation; the set of transition relations can be written as:

$T_{HSC(MD)} = \{\text{H_empty} \xrightarrow{\text{MD}} \text{H_occupied}, \text{H_occupied} \xrightarrow{\text{MD}} \text{H_occupied}\}$ and

$T_{HSC(NMD)} = \{\text{H_occupied} \xrightarrow{\text{NMD}} \text{H_empty}, \ \text{H_empty} \xrightarrow{\text{NMD}} \text{H_empty}\}$

4.2.2 Operational semantics for the composition of Home Automation components

Operational semantics, as presented in Section 3.3.2, are used for the composition of the Light Controller (LC) and the Home Status Controller (HSC). The LC and HSC components are designed to run in parallel and, from their initial states, the same event (MD) triggers a transition into their respective next states. Hence, according to the inference rule for the parallel composition presented in Section 3.3.2, the components can be composed as follows. For the first event, MD, the two possible transitions are as follows:

$$\frac{\text{AL_on} \xrightarrow{\text{MD}} \text{LBP}, \text{H_empty} \xrightarrow{\text{MD}} \text{H_occupied}}{\text{AL_on} \,||\, \text{H_empty} \xrightarrow{\text{MD}} \text{LBP} \,||\, \text{H_occupied}}$$

$$\frac{\text{LBP} \xrightarrow{\text{MD}} \text{LBP}, \text{H_occupied} \xrightarrow{\text{MD}} \text{H_occupied}}{\text{LBP} \,||\, \text{H_occupied} \xrightarrow{\text{MD}} \text{LBP} \,||\, \text{H_occupied}}$$

Similarly, for the second event, NMD, the two possible transitions are:

$$\frac{\text{LBP} \xrightarrow{\text{NMD}} \text{AL_on}, \text{H_occupied} \xrightarrow{\text{NMD}} \text{H_empty}}{\text{LBP} \,||\, \text{H_occupied} \xrightarrow{\text{NMD}} \text{AL_on} \,||\, \text{H_empty}}$$

$$\frac{\text{AL_on} \xrightarrow{\text{NMD}} \text{AL_on}, \text{H_empty} \xrightarrow{\text{NMD}} \text{H_empty}}{\text{AL_on} \,||\, \text{H_empty} \xrightarrow{\text{NMD}} \text{AL_on} \,||\, \text{H_empty}}$$

Since the initial start states were specified as AL_on and H_empty, and since MD and NMD are both events on which synchronisation occurs, the operational semantics ensure that there is no way of the composition entering other combinations of states such as AL_on || H_occupied and LBP || H_empty.

4.3 Introducing Fault Tolerance into the Home Automation System

Three different combinations of design diversity fault tolerance features are considered here for implementing fault tolerance: firstly the incorporation of an Acceptance Test mechanism with checkpointing in a Sequential execution scheme, secondly a Parallel execution with a Voter mechanism and thirdly a Parallel execution with a Hybrid voter (voter and acceptance test). These three design diversity software fault tolerance techniques are considered as basic techniques to cope with the design faults in operational system. These fault tolerance techniques will be considered in turn below in the context of the Home Automation System.

4.3.1 Applying an Acceptance Test Fault tolerance formalism

Using the style of operational semantics and handling of fault tolerance 'generics' presented in Section 3.3.2 and 3.3.4, an acceptance test can be applied to the LC and HSC components. This acceptance test can be represented as follows:

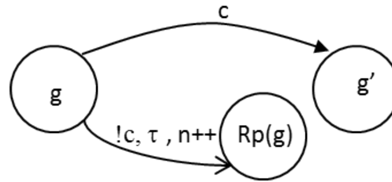


Figure 4.4: A state machine representing Acceptance Test mechanism and checkpointing

As presented in Section 3.3.4, the labels g and g' represent generic states and 'c' is the condition required for the acceptance test to be passed. The operational semantics can be applied here to compose this acceptance test with the LC component of Home Automation as follows, i.e. the composition of figure 4.2a and figure 4.4:

$$\frac{\text{AL_on} \xrightarrow{\text{MD}} \text{LBP}\ ,\ \text{g} \xrightarrow{\text{c}} \text{g}'}{\text{AL_on}||\text{g} \xrightarrow{\text{c,MD}} \text{LBP}} \quad \text{(i) acceptance test succeeds}$$

According to this inference rule, if the Acceptance Test is passed (c is true), then the state of the LC component progresses as usual ($\text{AL_on} \xrightarrow{\text{c,MD}} \text{LBP}$).

Similarly for the other transitions, if the acceptance test succeeds:

$$\text{g} \xrightarrow{\text{c}} \text{g}', \text{LBP} \xrightarrow{\text{NMD}} \text{AL_on} \Rightarrow \text{LBP} \xrightarrow{\text{c,NMD}} \text{AL_on}$$

$$\text{g} \xrightarrow{\text{c}} \text{g}', \text{AL_on} \xrightarrow{\text{NMD}} \text{AL_on} \Rightarrow \text{AL_on} \xrightarrow{\text{c,NMD}} \text{AL_on}$$

$$\text{g} \xrightarrow{\text{c}} \text{g}', \text{LBP} \xrightarrow{\text{MD}} \text{LBP} \Rightarrow \text{LBP} \xrightarrow{\text{c,MD}} \text{LBP}$$

In contrast, suppose the Acceptance Test fails:

$$\frac{g \xrightarrow{!c,\tau,n++} Rp(g)}{AL_on \| g \xrightarrow{!c,\tau,n++} Rp(AL_on)} \quad \text{(ii) acceptance test fails}$$

According to this inference rule, the Acceptance Test is not passed (c is false), and the state of the LC component is saved as a recovery point ($AL_on \xrightarrow{!c,\tau,n++} Rp(AL_on)$). This recovery point of the state is normally equivalent to the previous state.[1]

Similarly for the LBP state, if the acceptance test fails:

$$g \xrightarrow{!c,\tau,n++} Rp(g) \Rightarrow LBP || g \xrightarrow{!c,\tau,n++} Rp(LBP)$$

These rules can also be expressed with processes, as in inference rule (iii) of Section 3.3.4:

$$\frac{LC_i \xrightarrow{e} LC_i' , F \xrightarrow{c} F'}{LC_i \| F \xrightarrow{c,e} LC_i'} \quad \text{(iii) acceptance test succeeds}$$

If the Acceptance Test is passed (c is true), then process LC_i progresses as usual by transitioning to process LC_{i+1}'.

However, in the case where the Acceptance Test fails:

$$\frac{F \xrightarrow{!c,\tau} Rp(F)}{LC_i \parallel F \xrightarrow{!c,\tau} Rp(LC_i)} \quad \text{(iv) acceptance test fails}$$

However, the next version of the LC component is invoked with the inference rule (v) introduced in Section 3.3.4.

$$\frac{\exists LC_{i+1}}{(Rp(LC_i) \gg LC_{i+1}) \xrightarrow{\tau} LC_{i+1}[Rp(LC_i)] \parallel F} \quad \text{(v) acceptance test fails with versions}$$

This inference rule shows the sequential execution of n versions of LC, when the acceptance test fails. If version LC_i has transitioned into a recovery point in the case of a failed acceptance test, then the second version LC_{i+1} must be instantiated via sequential composition. The next version LC_{i+1} will be started at the saved recovery point represented by Rp (LC_i), and will continue execution in parallel with the fault tolerance component F.

---

[1] In this simple system, all states are saved as recovery points as there are only two states. In the case of more complex systems, only the latest states are saved and recovered on failure. On successful completion, these states are removed/deleted from the memory.

In the case that no more versions of the component exist i.e. LC_{i+1}, does not exist, the composition will be stopped.

$$\frac{\nexists \mathrm{LC}_{i+1}}{\mathrm{Rp}(\mathrm{LC}_i) \overset{\tau}{\rightarrow} \text{ stop}} \quad \text{(vi) no versions left}$$

Similarly, these rules can be applied to the composition of the fault tolerance mechanism (using generics) with the HSC component of Home Automation.

4.4.2 Applying a Fault tolerance formalism for Parallel Execution with Voter

Many fault tolerance mechanisms use parallel execution as an execution scheme. Voting is an important feature in parallel execution and is used in N-version Programming mechanisms. The Voter acts as an adjudicator and is responsible for checking the correctness of the result produced by n variants running in parallel. In case there is a majority agreement, the agreed result is used as the "correct" result. If there is no majority, failure in the result of the voter occurs; in this case, the state will either remain unchanged or the system will transition forward to a new safe state. The decision as to which approach will be taken will be specified in the configuration file of the Lex/Yacc pre-processor.

A state machine can be used to depict the behaviour of the Voter mechanism. In this state machine,

> let g represent any state from the system component (a ‘generic’ initial state),
>
> let g’ represent the future state after voting is applied,
>
> let r represent the result of voter
>
> let e be an event from the component with which the result of the voting will be composed.
>
> Let Rp(g) represent the saved recovery point of the generic state g, used in the case of the backward recovery,
>
> Let τ represent an internal event for transitions on failure to the saved recovery point where Rp(g) refers to the recovery point of the safe state g,

Graphically the voting mechanism can be represented as follows:

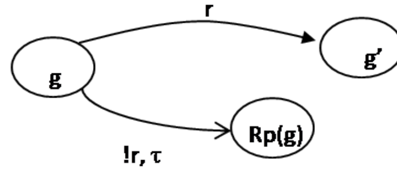


Figure 4.5: A state machine representing voting mechanism with backward error recovery

Note: there is no need to increment the version variable, n++, since all versions are executing in parallel.

The above finite state machine represents a voting mechanism with backward error recovery and needs to be composed with the components of the base system. The choice of forward error recovery and backward error recovery needs to be provided in the configuration files of the pre-processor. The rules for the composition of generic states with underlying system states have been presented in Section 3.3.4.

Applying these rules, g is a finite set of states which are considered as composition points, r is the voting result, g' is the next state after applying voting if voting is successful. In the case of !r, the state transitions to a safe state, that will have been saved as a recovery point in case of backward error recovery.

$$g \xrightarrow{r} g' \Rightarrow s \xrightarrow{r,e} s'$$

$$g \xrightarrow{!r,\tau} Rp(g) \Rightarrow s \xrightarrow{!r,\tau} Rp(s)$$

If, in contrast, forward error recovery is selected and voting fails, then this involves a transition to a new forward safe state represented as exc(s).

Fig 4.6 shows the voting mechanism with forward error recovery with the introduction of a new forward safe state 'exc(g)' as presented in Section 3.3.7.

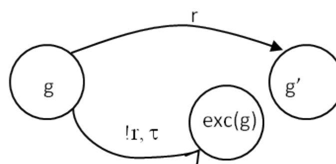


Figure 4.6: A state machine representing voting mechanism with forward error recovery

In case of forward error recovery, the system state transitions to a new safe state represented by exception handling, exc(g); this state will be mapped on to a new state from the underlying system component.

$$g \xrightarrow{!r,\tau} exc(g) \Rightarrow s \xrightarrow{!r,\tau} exc(s)$$

This composition is based on the following operational semantics and composition rules as presented in Section 3.3.2:

$$\frac{s \xrightarrow{e} s' \,,\, g \xrightarrow{r} g'}{s \,\|g \xrightarrow{r,e} s'} \quad \text{(i) Voting succeeds; r is the voting result}$$

In this first inference rule, if the voting is performed and r is the result, then the state of the underlying system component progresses as usual with an extra guard condition r.

In the second inference rule, if the voting is unsuccessful and fails to produce the majority result, then in the case of backward error recovery the state of the underlying system component moves into the saved recovery state:

$$\frac{g \xrightarrow{!r,\tau} \mathrm{Rp(g)}}{s \,\|g \xrightarrow{!r,\tau} \mathrm{Rp}(s)} \quad \text{(iia) Voting fails (backward error recovery)}$$

However, in case of voting failure and forward error recovery, a new safe state represented by exception handling will be introduced:

$$\frac{g \xrightarrow{!r,\tau} \mathrm{exc(g)}}{s \,\|g \xrightarrow{!r,\tau} \mathrm{exc}(s)} \quad \text{(iib) Voting fails (forward error recovery)}$$

Using the style of operational semantics presented above, the inference rule for composing the generics with the LC component of Home Automation can be applied as follows:

$$g \xrightarrow{r} g', \mathrm{AL_on} \xrightarrow{\mathrm{MD}} \mathrm{LBP} \Rightarrow \mathrm{AL_on} \xrightarrow{\mathrm{r,MD}} \mathrm{LBP}$$
$$g \xrightarrow{r} g', \mathrm{LBP} \xrightarrow{\mathrm{NMD}} \mathrm{AL_on} \Rightarrow \mathrm{LBP} \xrightarrow{\mathrm{r,NMD}} \mathrm{AL_on}$$
$$g \xrightarrow{r} g', \mathrm{AL_on} \xrightarrow{\mathrm{NMD}} \mathrm{AL_on} \Rightarrow \mathrm{AL_on} \xrightarrow{\mathrm{r,NMD}} \mathrm{AL_on}$$
$$g \xrightarrow{r} g', \mathrm{LBP} \xrightarrow{\mathrm{MD}} \mathrm{LBP} \Rightarrow \mathrm{LBP} \xrightarrow{\mathrm{r,MD}} \mathrm{LBP}$$

In the case of failure of voting and backward error recovery,

$$g \xrightarrow{!r,\tau} \mathrm{Rp(g)}, \mathrm{AL_on} \xrightarrow{\mathrm{MD}} \mathrm{LBP} \Rightarrow \mathrm{AL_on} \xrightarrow{!r,\tau} \mathrm{Rp(AL_on)}$$

In the case of forward error recovery,

$$g \xrightarrow{!r,\tau} \mathrm{Rp(g)}, \mathrm{AL_on} \xrightarrow{\mathrm{MD}} \mathrm{LBP} \Rightarrow \mathrm{AL_on} \xrightarrow{!r,\tau} \mathrm{exc(AL_on)}$$

According to the inference rules with processes presented in Section 3.3.2, the LC component can be composed with a fault tolerance feature as follows:

$$\frac{F \overset{r}{\rightarrow} F'}{LC \parallel F \overset{r,e}{\rightarrow} LC'} \quad \text{(iii) Voting succeeds}$$

If r is the voting result, then process LC progresses as usual and transitions to process LC'.

However, in the case where the voting does not produce the correct result:

$$\frac{F \overset{!r,\tau}{\longrightarrow} Rp(F)}{LC \parallel F \overset{!r,\tau}{\longrightarrow} Rp(LC)} \quad \text{(iv) Voting result with recovery point}$$

In the case of forward error recovery:

$$\frac{F \overset{!r}{\rightarrow} F''}{LC \parallel F \overset{!r,\tau}{\longrightarrow} s_3} \quad \text{(v) Voting result with forward recovery}$$

Similarly voting can be applied to the HSC component of Home Automation.

4.4.3 Applying the Fault tolerance formalism for the Hybrid Voter

The Hybrid Voter uses a combination of both the acceptance test and the voting algorithms presented above. In this mechanism, the result of different variants is first evaluated against acceptance test conditions c, and only accepted results are sent to the voter and generate the voting result r. The only difference with the acceptance test presented in Section 4.3.1 is the use of n versions. In the case of the Hybrid voter, all n versions are executed simultaneously in parallel. The results from the versions that have passed the acceptance test go forward for voting.

Using the same style of operational semantics, it is possible to give the following inference rules according to the possible operations presented above:

$$\frac{s \overset{e}{\rightarrow} s', g \overset{c\&\&r}{\longrightarrow} g'}{s \parallel g \overset{c\&\&r,e}{\longrightarrow} s'} \quad \text{(i) c is the acceptance test condition and r is the voting result}$$

In this first inference rule, if the acceptance test condition c is true and voting is performed and r is the result, then the state of the underlying system component progresses as usual with an extra guard condition c&&r before the other state starts.

A further possibility with the hybrid voting scheme is that the acceptance test condition passes, but voting fails to product the correct result. In this case, the state of the system component transitions to a new safe state called an exceptional state and represented as

exc(state); this needs to be defined in the configuration file of Lex and Yacc (as represented in Section 3.3.7).

$$\frac{g \xrightarrow{c\&\&!r,\tau} exc(g)}{s \,||g \xrightarrow{c\&\&!r,\tau} exc(s)} \quad \text{(ii) voting fails}$$

Using the style of operational semantics presented above, the inference rules can be applied to the LC and HSC components of the Home Automation system.

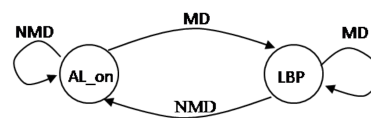
However, in the case of a hybrid voter mechanism, where multiple versions are executing in parallel, there is a chance for execution to be blocked. In this fault tolerance mechanism, the results of different variants (suppose v_1, v_2 and v_3) are first evaluated against independent acceptance test conditions c_1, c_2 and c_3. Importantly, only accepted results are sent to the voter to generate the voting result r. However, if the strictly synchronous operational semantics is applied to all the versions, then none of these versions can pass the composition criteria $c_1\&\&c_2\&\&c_3$, and execution will be blocked and will not progress to the voting. In this case, a refinement is required to the existing operational semantics to accommodate this. For example, in the case of hybrid fault tolerance, the conjunction operator forces each acceptance test to be true. This is too strong and, instead, all successful acceptance tests should go forward to voting, with any results that fail the acceptance test being ignored.

4.4.4 Pre-Processor Tool (Lex & Yacc)

The Pre-processor is written in Lex and Yacc and is used to handle fault tolerance generics and compose them with the base system components. The detail of handling these fault tolerance generics was presented in Section 3.3.4, and can be applied to the LC and HSC components as follows:

Spec: LC.txt

```
Define Comp;
states = {AL_on, LBP};
events = {MD, NMD};
start  = AL_on, MD; // needs to specify the
initial state and initial event for NuSMV
transitions = {
            AL_on, _, MD, _, LBP;
            LBP, _, NMD, _, AL_on;
            AL_on, _, NMD, _, AL_on;
            LBP, _, MD, _, LBP};
```

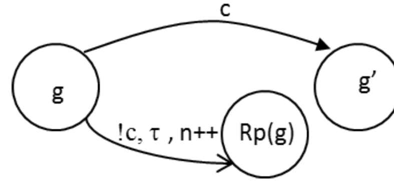


The specification file for the acceptance test is as follows:

Spec: at.txt

```
Define FT;
states = {g, g'};
condition = {c};
n=1; // number of software version
```

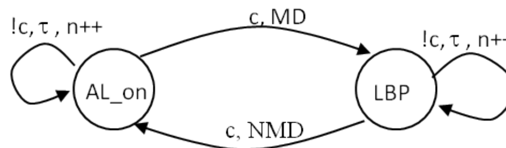
```
max_n =3; // max number of versions
start  = g, c;
transitions = {
        g, c, _, _, g';
        g, !c, _, n++, Rp(g);};
```



According to the operational semantics presented in Section 4.2 and 4.3, the pre-processor performs the composition of the LC component and the fault tolerance acceptance test. An algorithm for the composition of system components with the fault tolerance features has been given in pseudocode in Section 3.3.7. According to this algorithm, in Yacc, the generic states of the AT are mapped to the states of the LC component with the additional guard condition 'c' for the acceptance test. The following basic composition is obtained from the Lex and Yacc pre-processor:

$$\mathrm{g} \xrightarrow{\mathrm{c}} \mathrm{g}', AL_on \xrightarrow{\mathrm{MD}} LBP \Rightarrow AL_on \xrightarrow{\mathrm{c,MD}} LBP$$
$$\mathrm{g} \xrightarrow{\mathrm{!c,\tau,n++}} \mathrm{Rp(g)}, AL_on \xrightarrow{\mathrm{MD,n=1}} LBP \Rightarrow AL_on \xrightarrow{\mathrm{!c,\tau,n++}} AL_on$$
$$\mathrm{g} \xrightarrow{\mathrm{c}} \mathrm{g}', LBP \xrightarrow{\mathrm{NMD}} AL_on \Rightarrow LBP \xrightarrow{\mathrm{c,NMD}} AL_on$$
$$\mathrm{g} \xrightarrow{\mathrm{!c,\tau,n++}} \mathrm{Rp(g)}, LBP \xrightarrow{\mathrm{NMD,n=1}} AL_{on} \Rightarrow LBP \xrightarrow{\mathrm{!c,\tau,n++}} LBP$$

The graphical representation of this composition is as follows:



Similarly, generics are applied to HSC and their composition. The detailed code of Lex and Yacc with the generated input file for NuSMV is presented in Appendix A.

4.4.5 Lex and Yacc generated NuSMV Models for Home Automation Components

The above section has shown how Lex and Yacc can be used to compose different FT mechanisms with the underlying system components. It is now necessary to compose these composite FSMs together in order to analyse the overall behaviour using NuSMV.

The finite state machines presented below show the graphical representation of the NuSMV model presented in Appendix A for the Light Controller and Home Status Controller composed with fault tolerance mechanisms, as generated by the pre-processor.

For the Home Automation components, this principle is used to introduce fault tolerance into the system. Suppose that 3 variants of the Light Controller (LC) component are running sequentially. Let process P_1, P_2, P_3 represent the behaviour of LC component versions 1, 2 and 3, the states of the primary process P_1 are saved as recovery points. If the acceptance test evaluation condition 'c' is passed, the process's state is progressed as usual; otherwise if the test fails or if any errors are detected by other means, then the system is recovered back to the latest safe recovery point 'Rp' and the second process P_2 starts executing from that particular Rp, as illustrated in Figures 4.7a and 4.7b below.

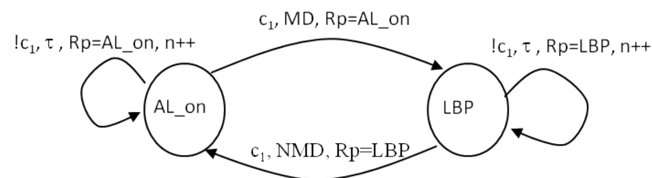


Figure 4.7a: Composite FSM of Light Controller (LC) with AT

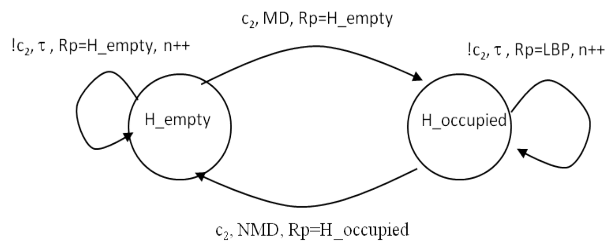


Figure 4.7b: Composite FSM of Home Status Controller (HSC) with AT

Key: c: Acceptance Test Condition, Rp = Recovery Point, MD: Motion Detection, NMD: No Motion Detection, LBP: Lighting by Presence, AL_on: Automated Lights On, H_empty: Home Empty, H_occupied: Home Occupied, τ: internal action

When composed together, it is expected that, if the home is empty, then the *Automated Lights* feature will be enabled; otherwise the *Lighting by Presence* feature will be utilized. The Home Status Controller indicates whether the status of the home is empty or occupied. On the basis of this information, the Illumination component enables the feature of *Automatic Light On* or *Lighting by Presence*.

Through the analysis of traces in NuSMV, the following composed FSM is generated. This represents the composition of the above two composite FSMs. Figure 4.8 below shows the automatic composition of these two state machines in NuSMV

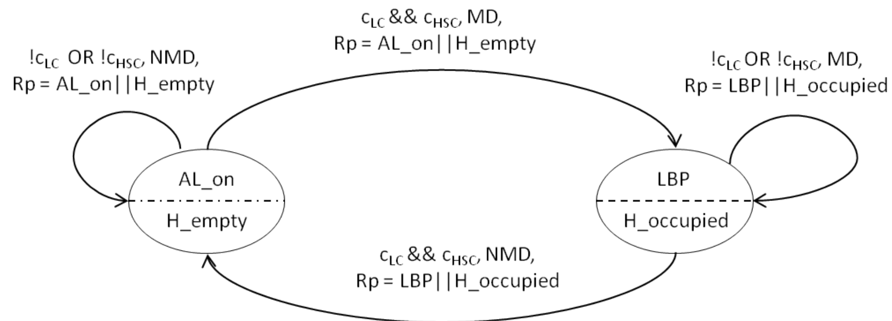


Figure 4.8: Composed Fault Tolerant Home Automation State Machine

4.5 Feature Interaction Analysis

The above finite state machines of LC and HSC components of a Home Automation as well as fault tolerance features have been generated by the Lex and Yacc pre-processor, and are presented (in textual form) in Appendix A. The Lex and Yacc pre-processor generates the files for the input language of NuSMV model checking. The input language of NuSMV is composed of variable declarations, state initialisations, state transitions, and a list of properties written in temporal logic formulae. In NuSMV, LTL specifications are introduced by the keyword "LTLSPEC", whereas CTL specifications are introduced by the keyword "SPEC".

With the introduction of fault tolerance mechanisms (acceptance tests) into two components of Home Automation, one area of potential feature interaction to check for is, if one component proceeds normally, whilst the other one fails. This may potentially result in inconsistent states such as LBP||H_empty or AL_on||H_occupied.

However, the operational semantics presented in Section 3.3.5 force synchronisation on the shared events, MD and NMD, so these inconsistent states can never occur.
Graphically these potential feature interactions can be represented as in Figure 4.9 below.

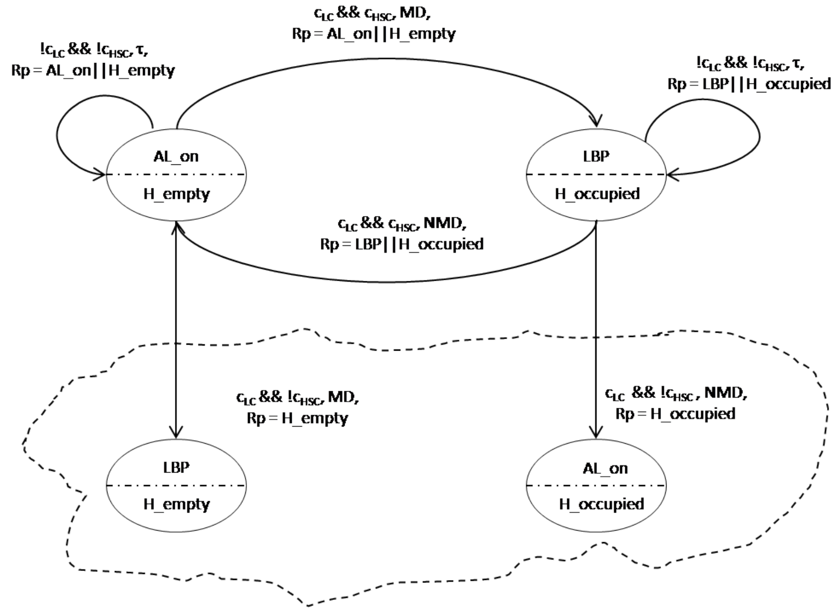


Figure 4.9: Potential Feature Interactions

Beyond this specific case, further potential feature interactions can be validated through model checking logic properties as follows.

Assertion Invalidation

a) SPEC AG !(LC.state = AL_on & HSC.state = H_occupied)

This model checking formula indicates that if the state of the light controller is Automatic Lights on (AL_on) then there will never be the case where the status of the Home is Home Occupied (H_occupied). In Figure 4.8 it is clearly shown that with the recovery points and acceptance test, the states of the Light Controller (LC) or the Home Status Controller (HSC) synchronise on the event MD or NMD, and never lead to an inconsistent state. The formula is true as there is no such case where Home is occupied and the Automatic light mode is on, and this property is shown to be valid through model checking with the tool NuSMV.

b) SPEC AG!(LC.state = LBP & HSC.state = H_empty)

Similarly, this model checking formula indicates that if the state of the light controller is lighting by Presence then there will never be the case where simultaneously the status of the Home is empty. Again, this formula is true and is validated through model checking with NuSMV.

Fault Tolerance Property

a) SPEC AG((LC.state = AL_on &!c_{LC}) -> AX(LC.state = AL_on))

This formula states that if LC is in the AL_on state and the fault tolerance acceptance test condition is false, then the state will remain unchanged and acts as a recovery point.

b) SPEC AG((LC.state = LBP & !c_{LC}) -> AX(LC.state = LBP))

Similarly, this formula states that if LC is in the LBP state and acceptance test condition is false then the state will remain unchanged.

Both of these logic formulae can be verified by model checking through NuSMV.

4.6 Introducing New Features in Home Automation

In the feature diagram for the smart home in Figure 4.1, an Alarm feature is present under the Security feature. For the illustrative purpose of this feature interaction analysis, suppose that the Alarm feature is enabled in case of 'Away from Home' functionality. The purpose of the Alarm feature is to detect motion and any suspicious activity in the home results in the alarm being turned on.

Secondly, the 'Away from Home' functionality also aims to give the impression that the house is occupied by enabling Automatic Lights on at random or pre-defined times.

Finally, a new feature of Climate Control can also be added to the Home Automation. The Climate Control feature is represented as an optional feature in feature diagram presented in Figure 4.1. The Climate Control service controls the heating of the house, the air-conditioning, the window blinds and also the windows.

Scenario. Imagine the situation in the case that a home's occupants are away from home. The owner sets the Alarm feature and Automatic Lights feature on to protect home from intruders. The Alarm as well as Light Controller monitors the state of the house through motion detection sensors in the house. In addition to motion detection, the Alarm feature also monitors the other objects like lights, blinds, doors, etc. The owner has also turned on the climate control feature to control the heating of the house, and has the capability to achieve this by closing/ opening blinds and windows.

In this scenario, when the automatic lights are turned on, the alarm will be triggered resulting in a call out. Similarly, when the climate control feature opens a window, again the Alarm feature will be triggered. This behaviour is undesirable and can be classified as a sequential

action interaction as described in Section 3.4. In this interaction, the climate control component opens the windows to lower the temperature, whereas, the motion sensor detects movement in the environment that triggers the Alarm feature. Figure 4.10 shows this interaction graphically.

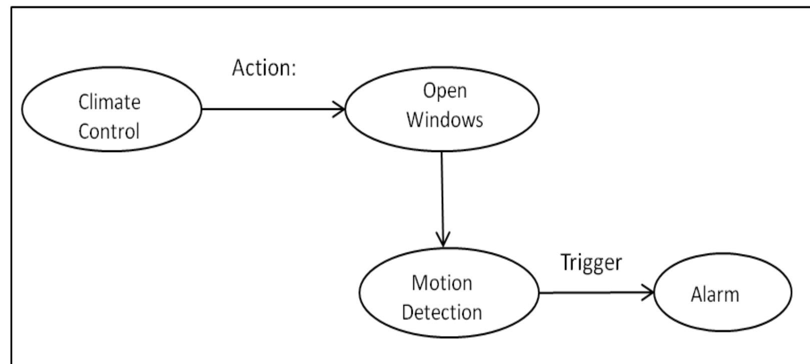


Figure 4.10: Sequential Action Trigger

The temporal logic for detecting this type of interaction is as follows:

SPEC AG !((ClimateControl.state = Window_open & MD) -> AF(Alarm_on))

This formula indicates that if the Climate Controller requests the action to open the Windows then there will never be the case where the sensor detects this motion. As represented in Figure 4.10, it is clear that this formula is not true and model-checking fails on this formula.

Similarly, a further shared trigger interaction can occur where the climate control service opens the windows and blinds and triggers the motion detection. Since motion detection is shared between different components of Home Automation, the movement caused by the climate control service can also trigger the Light Controller to move into the LBP state, the Home Status Controller to move into the H_occupied state, and also turns the security Alarm on. Figure 4.11 shows how motion detection triggers these other components.

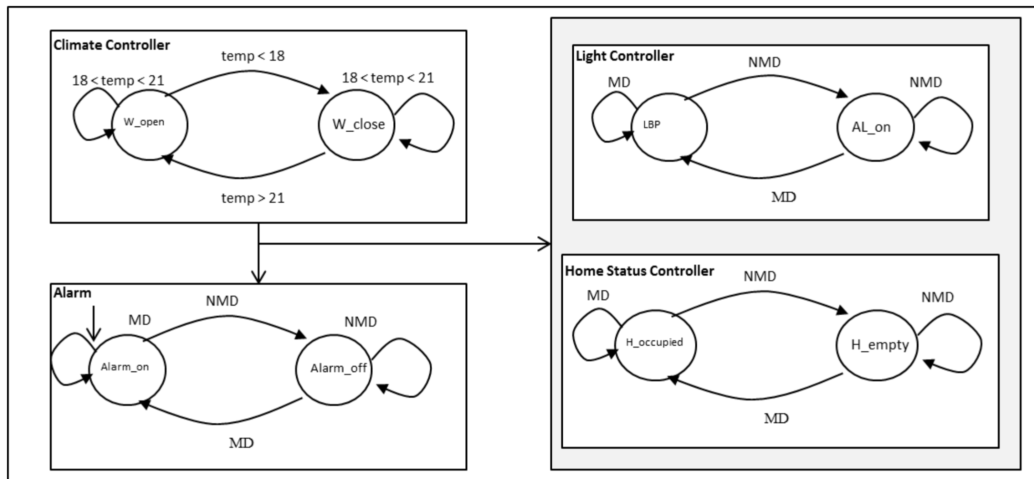


Figure 4.11: Climate Controller triggers the Alarm Component, Light Controller and Home Status Controller

The temporal logic for detecting this type of interaction is as follows:

SPEC AG (MD -> !AF(Alarm.state = on & (LC.state = LBP & HSC.state = H_occupied)))

This model checking formula checks whether the motion detection triggered by climate control service further invokes the other components. This formula is false as motion detection turns the alarm on and brings the light controller to lighting by presence mode, as represented in Fig 4.12.

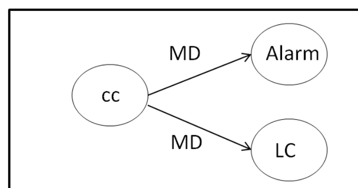


Figure 4.12: Shared trigger interaction between Climate Controller, Alarm and Light Controller

With the help of model checking verification, these types of undesirable interactions are detected that lead the system to inconsistent states. Further discussion and analysis will be presented in Chapter 6.

4.7 Summary

In this Chapter, the proposed orthogonal framework for the fault tolerance composition has been applied to the components of the Home Automation based on the operational semantics presented in Chapter 3. Using generics and the associated algorithm for the processing of generics, the Lex and Yacc pre-processor generates the input language for the NuSMV model checking tool. For the detection of feature interactions, CTL/LTL logic

properties are applied to the NuSMV model to verify the classified properties for the composed components of the Home Automation.

In summary, the ability of the Orthogonal Fault Tolerance Framework to provide fault tolerance generics based on the constraints and relationships is shown with the worked example of Home Automation. This Chapter has successfully shown the composability of orthogonal fault tolerance features and exhibited the feature interaction approach to detect potential undesirable feature interactions as classified.

Further evaluation and analysis will be presented in Chapter 6, based on the evaluation criteria of separation of concern, expressiveness of fault tolerance, composability, compositionality and soundness of feature interaction analysis.

In the following Chapter, a second case study will be considered, based on the Therac-25 system.

Chapter 5

Case Study: Therac-25

5.1 Introduction

The Therac-25 was a computer-controlled medical machine used for medical radiation therapy in the 1980s, and developed by AECL (Atomic Energy of Canada Ltd.). It was known as a linear accelerator, as it accelerated a beam of electrons at high energy to generate X-rays. Electron beams were used to treat shallow tissues whereas X-rays were used for deeper treatment.

The Therac-25 software was developed by a single person, using PDP 11 assembly language, and involved the re-use of some of the routines from the earlier Therac-6 and Therac-20 software. However, both of these earlier systems had a hardware single-pulse shutdown circuit; this acted as a hardware interlock to prevent overdosing by detecting an unsafe level of radiation and halting beam output after one pulse of high energy and current. In contrast, software was responsible for monitoring the whole procedure of the treatment with the Therac-25, including information about the prescribed dose, and setting up the machine by turning the beam on. When an operator turned the beam on, it was assumed that all the operational checks on the physical status of the machine were satisfied. This software was also responsible for the turning the beam off on the completion of the treatment, or on malfunctioning of the software. There was also an interlock system that was controlled by software, to shut down the machine in the presence of hardware malfunction.

The reason for selecting the Therac-25 as a case study is it has been well documented in terms of its behaviour, the various software errors that occurred, and the recommendations for future safety-critical software development. There are also several formal specification and verification studies available on this case study.

The Orthogonal Fault Tolerance (OFT) framework proposed in this thesis provides redundancy and self-checking capabilities through the introduction of design diversity fault tolerance features to reduce the chances of software errors. With the automatic

composition of these features based on the operational semantics, it is claimed to reduce serious software design errors. Hence the Therac-25 is a good candidate on which to evaluate the proposed OFT framework.

5.1.1 Accidents in Therac-25

There were six serious accidents between 1985 and 1987 due to overdoses of radiation by the Therac-25 system, in which three people were killed and three were seriously injured. The reasons behind these accidents were two-fold: software errors in the interface component and the failure of the software interlock system. These accidents have been well documented in [Leveson 1970][Leveson 1992][Turner 1993] and [Thomas 1994], but a summary is presented here as context for this case study.

- An Editing Problem in the Interface Component

The three most serious accidents happened due to two different software problems in the interface component of the Therac-25 software. The interface had many options to edit the patient's records to correct any data entry mistakes. Continuous pressing of the edit buttons and ignoring imprecise error messages such as "Malfunction 54" led, in two of the accidents, to the problem of an overdose being given to the patients. The machine also displayed a "treatment pause," indicating a problem of low priority in "dose input 2". There was no explanation of this error messages neither in a manual nor in documentation of the Therac-25. This was the cause of the third accident, since the operator had ignored the low priority warning (such warnings were commonplace and frequently ignored), and the system indicated that the full treatment dose had not yet been delivered; treatment continued to be administered and an overdose occurred.

- Failure in software interlock

The other accidents were due to a failure in the software that controlled the interlock of the Therac-25 machine. This problem was considered as a "failure in software interlock", where the light shield was not in a precise position. If the X-ray mode was selected, then the shield needed to be placed in between the beam and the patient. However, in these accidents, the shield position was not in the right place and this caused the radiation over-dose. The software interlock should have ensured that this did not happen, but the software failed to protect this critical state of the Therac-25 machine.

5.1.2 Recommendations

After investigating the reasons behind these incidents, it was found that the Therac-25 accidents involved software coding errors as well as errors in software requirements [Leveson 1993]. Basic software engineering principles such as proper documentation, software quality assurance, errors explanations, were violated in the Therac-25 software. The design of the Therac-25 software did not contain self-checks, error-detection and error handling features [Leveson 1995]. Some of these software errors had also existed in the Therac-20 software (elements of which had been re-used in the Therac-25 software), but independent hardware interlocks on this earlier machine had protected patients from radiation overdoses [Leveson 1993]. In contrast, the Therac-25 system relied primarily on software checks.

Further recommendations and regulations are recounted in [Jacky 1994] including:

- Introducing formal specification and model checking
- Redundancy and diversity should be incorporated
- Reducing the number of editing keys.
- Error messages explanations
- Documentation and manuals should be written properly so that reusability can be achieved.

Hardware related recommendations were also made such as design and testing of the hardware of the Therac-25. Suggested hardware changes were to have extra protection against software errors, where again the main focus was beam shutoff, energy mode and shield position. Hardware is beyond the focus of this thesis, but the proposed OFT framework will focus on the other recommended points from above.

5.1.3 Orthogonal Fault Tolerance and the Therac-25 System

As already stated in section 5.1.2, the Therac-25 accidents were due to poor software engineering principles when designing and implementing the system. There was no clear specifications and verification of the system, and the fault handling or fault detection mechanism embedded in the Therac-25 complex software were not sufficient to deal with design faults. It was also noted that the design of such a complex medical safety critical system was developed by a single programmer [Leveson and Turner 1993].

With reference to the recommendations suggested earlier, the proposed orthogonal fault tolerance framework offers diversity and redundancy along with the formal specifications of the system. The OFT framework also provides the composition mechanism to combine different design diversity fault tolerance features with the software system. Design diversity fault tolerance mechanisms provide features such as an adjudicator (acceptance test, voter, or hybrid voter) that can be used to verify the correctness of the behaviour. There is also a clear need to model the Therac-25 in a way that processes may be parameterised [Thomas 1994]. This parameterisation can be done by introducing different fault tolerance mechanisms, as will be illustrated in section 5.2.

With the help of the OFT framework, the composition of fault tolerance mechanisms with the system's components is carried out with the help of operational semantics and logic properties are verified with a model checking tool. Logic properties can be written in CTL or LTL to specify and verify the desired properties of the interface and the machine components. This will be illustrated in section 5.3. Finally, as will be shown in section 5.4, the proposed framework also provides a way to analyse undesirable interactions.

5.1.4 Features of the Therac-25 System

In this section, the features of the Therac-25 are presented in terms of the feature diagram and dependency relationship. In the Therac-25 system, there are two modes of operation for the radiation beam, one is the electron mode and the other is the X-ray mode. An electron beam (low energy) can be fired directly at a patient whereas, for the X-ray beam, a shield is placed and bombarded with accelerated electrons (high energy) that results in an X-ray beam. The risk here was that it proved possible to fire the high energy electrons (X-ray) at the patient directly without placing the shield correctly in between. Therefore, the position of the shield has a significant importance in the whole procedure of radiotherapy in the Therac-25.

The Therac-25 software has a number of main features as represented by the following feature diagram (Figure 5.1).

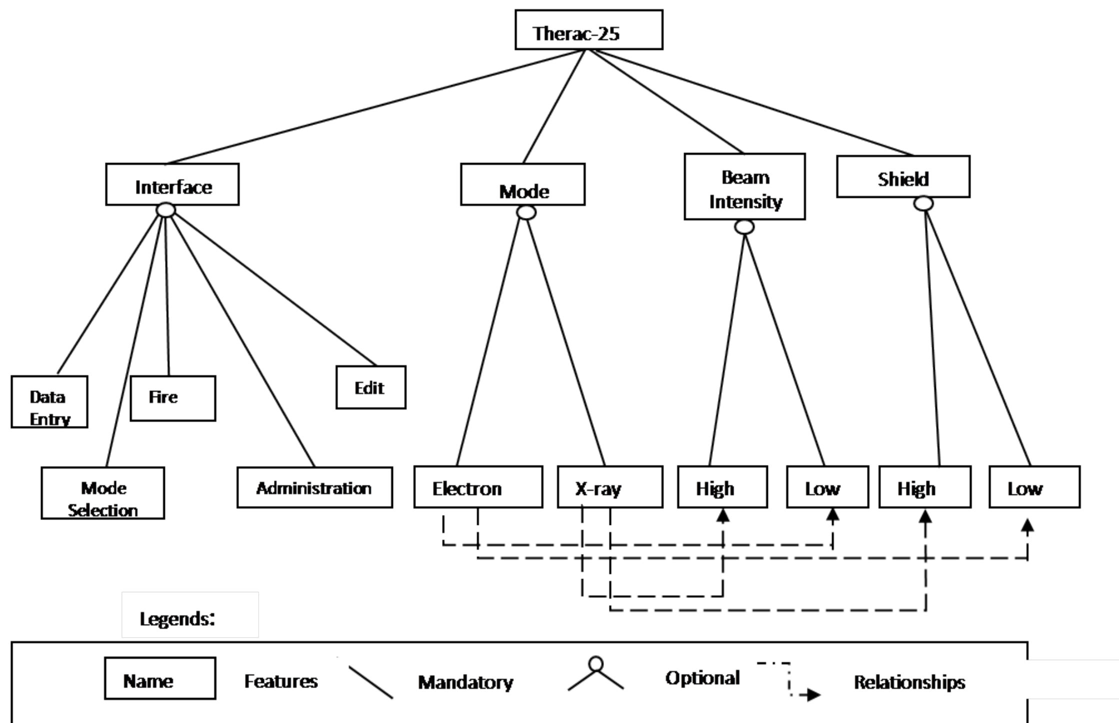


Figure 5.1: Therac-25 Software Features

In the Therac-25 software, there are a few mandatory features:

- Interface: the beam data can be entered by selecting the prescribed mode, patient data can be entered and edited, and the beam can be fired and the treatment can be administered;
- Mode of operation: the mode can be set to either X-ray treatment or electron treatment;
- Beam intensity: can be set to a high intensity beam (for X-ray treatment) or, by default, a low intensity beam (for electron treatment);
- Shield/Spreader position: can be set to a high position (for X-ray treatment) or, by default, a low position (for electron treatment);

5.2 Formal Specification of the Therac-25 System

Initially, in section 5.2.1, the Therac-25 system will be presented using Statecharts, as represented in [Bolton 2008]. The clarity of specification, and the similarity with the finite state machines required by the proposed OFT framework, mean that this published formal model is adopted as the representative specification of the Therac-25 system. Then in section 5.2.2, a final state machine representation of the Therac-25 system will be given, mirroring the Statechart specification, but in the format used in the OFT framework.

5.2.1 Statechart Model for the Therac-25 Components

In the Statechart model of the Therac-25 system [Bolton 2008], there are two main components of the Therac-25 software, namely:

- Interface Component (involved in the error in editing; see section 5.1.1)
- Machine Component (involved in the software interlock error; see section 5.1.1)

Interface Component of the Therac-25: Statechart

Figure 5.2 shows the Statechart model for the Therac-25's interface component [Bolton 2008]. At initialization, the interface is in the edit mode, and the mode of operation is in neither the x-ray or electron beam mode.

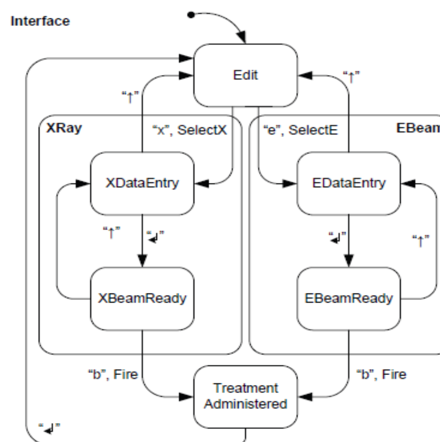


Figure 5.2 Therac-25 Interface Statechart [Bolton 2008]

Key: x: X-ray, e: Electron, b: Beam, ↲: Enter, ↑: Edit/Change

This statechart shows that from the initial Edit state, on the occurrence of event "SelectX", the system will go to the "XDataEntry" state. Similarly on the occurrence of "SelectE" event, the next state will be "EDataEntry". In states, "XDataEntry" and "EDataEntry", the parameters for the X-ray and Electron beams are entered and can be edited. The upward error shows that the editing leads the system to the edit state. An Enter event leads the system to the "XBeamReady" and "EBeamReady" state, and then the Fire event indicates that treatment is administered while the beam is firing.

The first accident mentioned in section 5.1.2 was caused due to an interface component design error. The process of editing a patient's data to correct an error allowed the machine to enter into an unsafe state whereby the machine administered X-rays to patients, without

the spreader (shield) in place[1]. This can be proved by model checking CTL formulae, but first it is necessary to present the Statechart model for the machine component since synchronisation occurs between the interface and machine components on the events SelectX/E and Fire.

Machine Component of the Therac-25: Statechart

Figure 5.3 shows the Statechart model for the Therac-25 machine component; this comprises of three Statecharts, namely BeamLevel, Spreader and BeamFire [Bolton 2008].

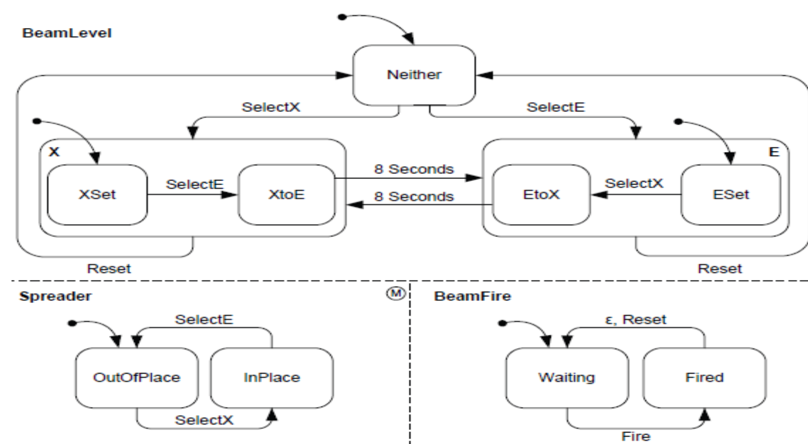


Figure 5.3 Therac-25 Machine Component Statechart

At initialisation, all three statechart models are synchronised awaiting SelectX and SelectE commands. In BeamLevel model, with the SelectX command, the system transitions to mode X (x-ray), whereas the SelectE command causes the system to transition to mode E (electron). The transition from mode X to mode E (or vice-versa) takes 8 seconds; in the meantime the Statechart resides in the intermediate states (XtoE or EtoX). A Reset command sets the machine back to the initial state.

For the spreader model, the SelectX command moves the spreader (shield) from out of place to in place. The SelectE command then moves the spreader back to out of place. When the Fire command is seen by the BeamFire model, it transitions into the Fired state, indicating that the beam has been fired. The machine then automatically resets to the initial state once the prescribed dose has been delivered.

[1] The terms spreader and shield are synonymous in the context of Therac-25. When the spreader is said to be in place, this is equivalent to the shield being in the high position, and vice-versa.

Model Checking the Unsafe Condition

As mentioned above, the unsafe condition caused by a software design error in the interface component can be verified by model checking. Bolton et al. evaluate this unsafe condition with the following CTL formula:

CTL#1: AG ¬(BeamLevel = X ∧ Spreader = OutOfPlace ∧ BeamFire = Fired)

This formula checks that in every future state, it is *not* the case that the system can be in the x-ray mode, the spreader (shield) be out of place, and the beam be fired.

As reported in [Bolton 2008], when model checking was performed on this formula, and applied to the Interface Statechart above, the condition was not satisfied; this was the reason behind one set of fatal accidents caused by the Therac-25 software.

5.2.2 Finite State Machine Model for the Therac-25 Components

For the proposed orthogonal fault tolerance framework, the Statechart components of the Therac-25 are first mapped on to finite state machine (FSM) representations.

Interface Component of the Therac-25: FSM

As above, the interface component is considered as the first component of the Therac-25 to study. The finite state machine representation of the Statechart interface component (Figure 5.2) is given in Figure 5.4 below.

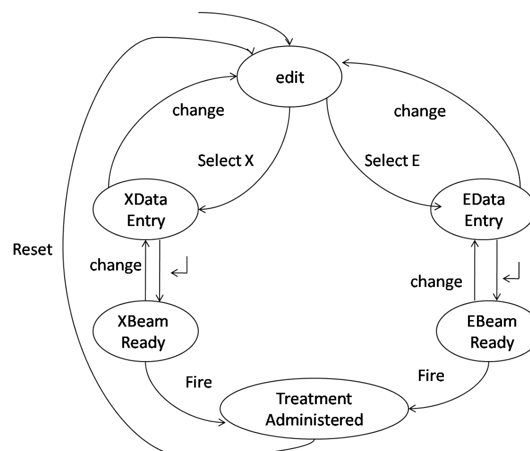


Figure 5.4 Therac-25 Interface Component

Key: x: X-ray, e: Electron, b: Beam, ↵: Enter, Change

The textual form of the finite state machine (in NuSMV format) is presented in Appendix B. Again, as with the Statechart models, before any logic formulae can be model checked, it is first necessary to present the FSM for the machine component.

Machine Component of the Therac-25: FSM

The second component is the machine component of the Therac-25. The earlier Statechart model has also been translated into a finite state machine representation. In the Statechart model, there were three sub-models, namely beam level, spreader and beam fire, all running in parallel. The finite state machine representation of the Statechart model of the machine component (Figure 5.3) is given in Figure 5.5 below. Note that the sub-model X from Figure 5.3 is represented by XSet and the sub-model E is represented by ESet and the transition is guarded with time of 8 sec.

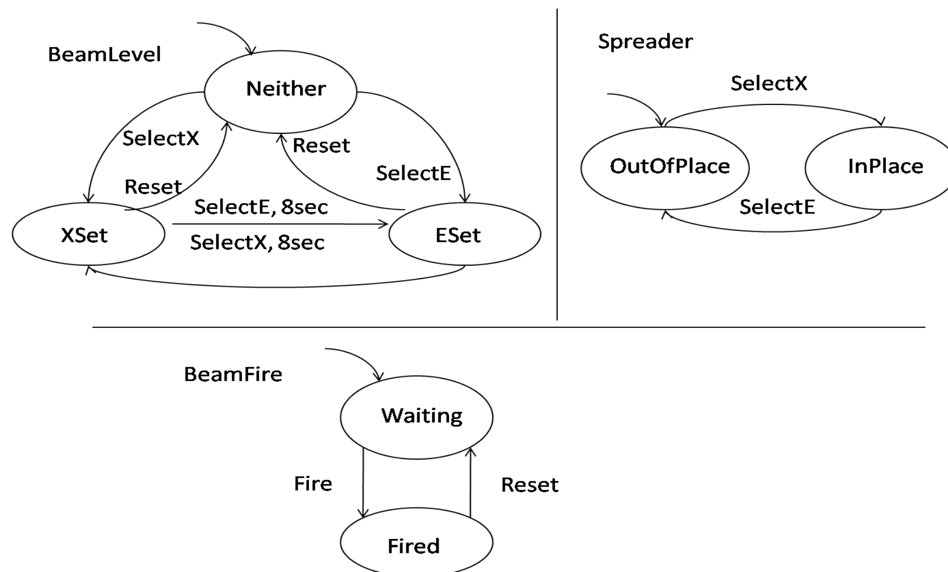


Figure 5.5: The Machine Component of the Therac-25

Again, the textual form of the finite state machine in NuSMV format is given in Appendix B.2.

Model Checking the Unsafe Condition

As with the Statechart model, the same logic property CTL#1 can be used to evaluate the unsafe condition of the Therac-25 system:

CTL#1: AG ¬(BeamLevel = X ∧ Spreader = OutOfPlace ∧ BeamFire = Fired)

This CTL formula ensures that, for all paths through the model, it will never be the case that the beam level is set to X-ray, the spreader is out of place and the beam is fired.

However, as expected, this formula is false for the NuSMV representation of our FSM model. A counterexample trace is provided by the NuSMV tool:

Countertrace#1:

```
-> State: 1.1 <-
        Interface.state = Edit
        BeamLevel.state = Neither
        Spreader.state = OutOfPlace
        BeamFire.state = waiting
        Command = SelectX
-> State: 1.2 <-
        Interface.state = XDataEntry
        BeamLevel.state = XSet
        Spreader.state = InPlace
        Command = Fire
-> State: 1.3 <-
        BeamFire.state = Fired
        Command = SelectE
-> State:1.4 <-
        Spreader.state = OutOfPlace
        Command = SelectX
```

As expected[2], this trace shows that the software of the Therac-25 is unsafe: in the last part of the trace, the command is for the X-ray beam when the spreader is still out of place.

5.2.3 Summary

This section has presented the implementation of the interface and the machine components as finite state machines in NuSMV. Traces have been provided to show the underlying unsafe conditions for the Therac-25 components; this replicates previous work from [Bolton et al. 2008] and [Thomas 1993].

However, as yet, there are no fault tolerance mechanisms incorporated in the specifications as required by the OFT framework. This is addressed in the following section.

5.3 Fault Tolerance Composition

In this section, design diversity fault tolerance mechanisms are composed with the machine the interface components of the Therac-25 with the help of Lex and Yacc pre-processor that generates the input file for NuSMV. The complete composition details can be found in

[2] This is consistent with both [Bolton et al. 2008] and [Thomas 1993] who identify an identical counter-trace.

Appendix B. The operational semantics for the composition of fault tolerance mechanisms to the components follow those already presented in Chapters 3 and 4. Logic properties written in CTL and LTL will be used to prove the correctness of the behaviour once the fault tolerance mechanisms have been introduced.

5.3.1 Composition of N-version Programming Mechanisms with the Therac-25 Component

As described in chapter 3, there are various different software fault tolerance mechanisms that could be applied to this case study, including Recovery block, N-version programming, and N-self checking programming mechanisms. Similarly, there are various different forms of adjudicator that can be used with these fault tolerance mechanisms. A few of these mechanisms have already been used with the worked example of the Home Automation system presented in Chapter 4, such as AT and Recovery block. For illustrative purposes here, the N-version programming mechanism is selected to compose with the Therac-25 system. The adjudicator that will be used with N-version programming mechanism is the "voter". The following constraint is relevant to N-version programming as presented in Section 3.2.2:

Versions running in parallel either use a voter or a hybrid adjudicator;

$$PAR \Leftrightarrow VOT \vee HYB.$$

In the context of the above constraint, the adjudicator "voting" is used for the parallel execution of 3 versions of software responsible for the computation of the Spreader position in the Therac-25 software. Firstly the position of the spreader/shield is manually set by the operator; this occurs before the system enters the prescribed data entry state (XDataEntry or EDataEntry). Secondly, the software then verifies the position with the sensor data; the finite state machine sub-model, *Spreader* in Fig 5.6, determines the automatic placement of the spreader based on the chosen event. According to this model, on the occurrence of the command *XSelect*, the spreader moves to *InPlace* position, while on the occurrence of the command *ESelect*, the Spreader comes back to its initial state as *OutOfPlace*.

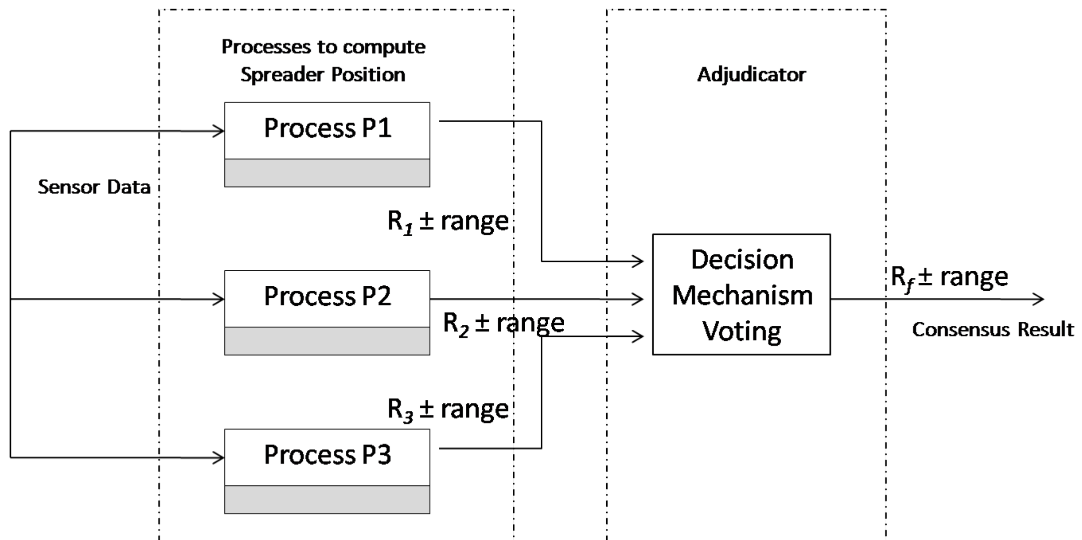


Figure 5.6: Component to compute sensor values for the spreader position with N-version Programming

The N-version programming model that uses a voter as adjudicator is presented in Fig 5.6. The three independent processes with different specifications, P1, P2 and P3, are executed in parallel and used to compute the position of the spreader based on the sensor data; the result of each individual process is denoted by R_i ± range (where range denotes a margin of error). The voter then acts as an adjudicator and uses a majority consensus to decide upon the final result, denoted by R_f. This final consensus result is composed with the finite state sub-model: Interface – where it is considered as a guard condition to transition from the Edit state to X/ EDataEntry states.

In a similar way to section 4.4.2, the voting mechanism can be applied to the Interface component of the Therac-25 software system and represented graphically as follows:

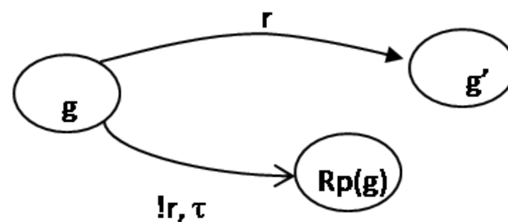


Figure 5.7: A state machine representing voting mechanism with backward error recovery

The above finite state machine represents the orthogonal voting mechanism with backward error recovery and needs to be composed with the Interface component of the Therac-25 machine from Fig 5.4.

As presented previously, g and g' represent generic states that are considered as composition points, and r is a condition that represents the result of the voter. In the case of !r, the state transitions to the saved recovery point, represented as Rp(g). This composition is based on the following operational semantics and composition rules already presented in Chapters 3 and 4.

$$\frac{s \overset{e}{\rightarrow} s', g \overset{r}{\rightarrow} g'}{s \parallel g \overset{r,e}{\rightarrow} s'} \qquad \text{(i) Voting succeeds; r is the voting result}$$

In this first inference rule, if the voting is performed and r is the result, then the state of the underlying system component progresses as usual with an extra guard condition r.

In the second inference rule, if the voting is unsuccessful and fails to produce a majority result, then in the case of backward error recovery the state of the underlying system component moves to the saved recovery point.

$$\frac{g \overset{!r,\tau}{\longrightarrow} Rp(g)}{s \parallel g \overset{!r,\tau}{\longrightarrow} Rp(s)} \qquad \text{(ii) Voting fails (backward error recovery)}$$

As will be presented below in Figure 5.8, the finite state machine of the Interface component of the Therac-25 is augmented with the additional guard condition as "SP = R_f", where SP is used for the position of the spreader computed by three versions of software based on the sensor data. 'R_f' refers to the final result achieved from the voting algorithm. This condition is used to guard the selection of the beam (X-ray or Electron).

In the state machine model, this means that the condition R_f guards the events SelectX and SelectE when transitioning from the Edit state to the states XDataEntry and EDataEntry. Hence, the possible execution options for the composition of fault tolerance with this portion of the Interface component are as follows:

$$\frac{\text{Edit} \xrightarrow{\text{SelectX}} \text{XDataEntry}, g \overset{R_f}{\rightarrow} g'}{\text{Edit} \parallel g \xrightarrow{R_f,\text{SelectX}} \text{XDataEntry}} \qquad \text{(ia) Voting succeeds; } R_f \text{ is the final voting result}$$

Similarly, for the event ESelect,

$$\frac{\text{Edit} \xrightarrow{\text{SelectE}} \text{EDataEntry}, g \overset{R_f}{\rightarrow} g'}{\text{Edit} \parallel g \xrightarrow{R_f,\text{SelectE}} \text{EDataEntry}} \qquad \text{(ib) Voting succeeds; } R_f \text{ is the final voting result}$$

In the second inference rule, if the voting is unsuccessful and fails to produce the majority result, then the state of the underlying system component remains unchanged and presents as follows:

$$\frac{g \xrightarrow{!r,\tau} g}{Edit \parallel g \xrightarrow{!r,\tau} Rp(Edit)} \qquad \text{(ii) Voting fails (backward error recovery)}$$

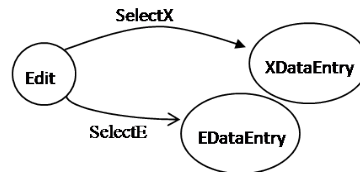
The complete implementation is provided in Appendix B.2.

5.3.2 Pre-Processor Tool (Lex & Yacc)

As presented in Chapter 3 and 4, the Pre-Processor written in Lex & Yacc is used to handle fault tolerance generics independently based on the algorithm presented in Section 3.3.5. This can be applied to the Interface component of the Therac-25 software as follows. Again, only the portion relevant to the fault tolerance mechanism and resulting guard is shown; however, the full state machine is shown in Figure 5.8 and is included in Appendix B.

Spec Therac.txt

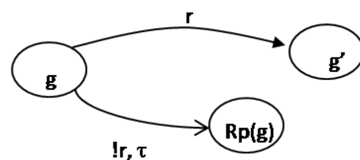
```
Comp: Interface
states = { Edit, DataEntry, DataEntry};
events = {SelectX, SelectE};
start = Edit
transitions = {
Edit, _, XSelect, _,  XDataEntry
Edit, _, ESelect, _, DataEntry};
```



The specification file for the voting mechanism is as follows:

Spec ft.txt

```
states = {g, g'};
condition = {r};
start = {g, r};
transitions = {
g, r, _, _, g';
g, !r,_, _, Rp(g)};
```



With the operational semantics and composition rules presented in Section 3.3.2 and 3.3.5, the pre-processor performs the composition of the Therac-25 components and the fault tolerance voting mechanism with backward recovery. The graphical representation of this composition is as follows:

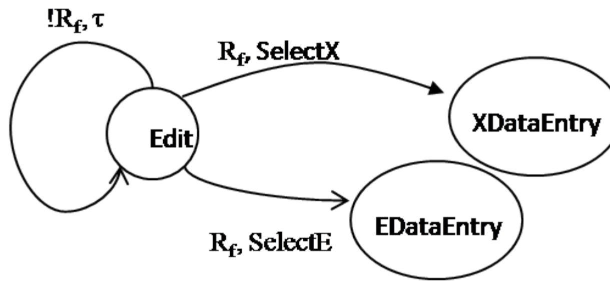


Figure 5.7: A finite state machine representing the result of voting mechanism composed with Interface component finite state machine.

Figure 5.8 presents the Lex and Yacc generated model for the Interface component, shown next to the other Therac-25 components.

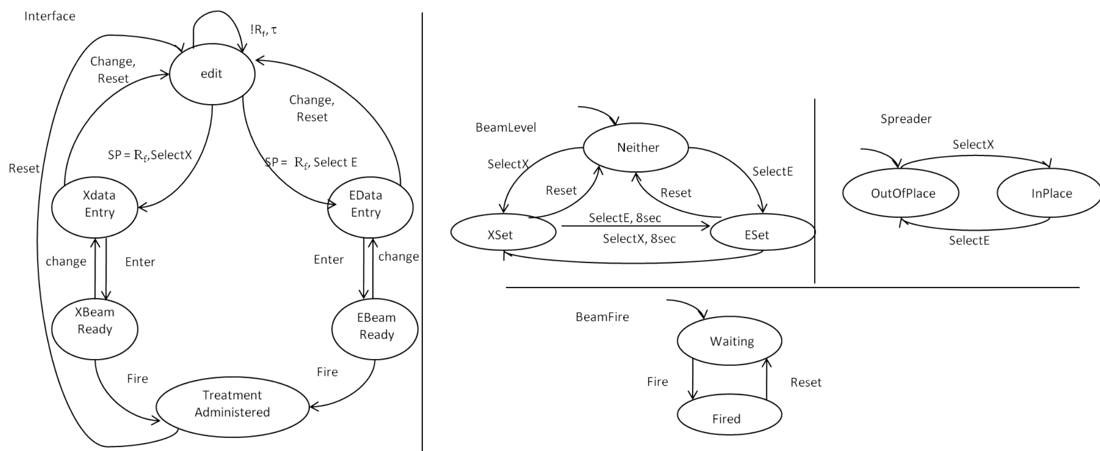


Figure 5.8 Interface Component with voting result for Spreader position

Keys: R_f: Voting result for Spreader Position, !R_f: unsuccessful voting result

5.4 Model Checking for the Safe Condition and Feature Interaction Analysis

By introducing the N-version programming fault tolerance mechanism to the Interface component of the Therac-25 machine, it is claimed to reduce or eliminate the chances of software design errors. Specifically, by using N-version programming to check the position of the spreader, the resulting guard condition should always be composed with the selection of the beam. With this composed fault tolerance, there are now two options:

(i) The voter succeeds and reaches a majority voting, that allows the X-ray or Electron beam to be selected and the component transitions to the respective data entry state.

(ii) The voter fails to reach a majority voting, in which case the component remains in the safe state until the operator manually changes the spreader position.

CTL and LTL logic properties are applied to the NuSMV model of the Therac-25 interface and the machine component (composed with the N-version programming mechanism for the automatic computation of the spreader position), to verify the safety of the model. The implementation detail of the Therac-25 component in NuSMV, the logic properties, and the NuSMV output are all given in Appendix B.2. The original CTL#1 logic property has been translated into the context of the different components in NuSMV as follows:

CTL#1: AG ¬(BeamLevel = X ∧ Spreader = OutOfPlace ∧ BeamFire = Fired)

Translated CTL#1: SPEC AG ! (BeamLevel.state = XSet & Interface.state = XBeamReady & Spreader.state = OutOfPlace & BeamFire.state = Fired)

This CTL formula states that for all paths through the model, it will never be the case that the beam level is set to X-ray, the spreader is out of place and the beam is fired.

When model-checking is applied to the formula, the safety property is found true as expected. This demonstrates that the composition of the fault tolerance mechanism has prevented the critical software error of the Therac-25 system. In the case that voting succeeds, the system composition continues as normal, whereas in the case that voting fails, the system remains in a safe state and awaits the operator to manually adjust the spreader position.

In a similar manner, NuSMV can also be used to prove the symmetric property where the system is in Electron mode but the spreader is InPlace:

CTL#2: SPEC AG ! (BeamLevel.state = ESet & Interface.state = EBeamReady & Spreader.state = InPlace & BeamFire.state = Fired)

Assertion Invalidation: The original Therac-25 error originated from the ability for the Beam level to be in ESet mode while the Interface component is in X-ray mode, and vice-versa.

These properties can be classified as assertion invalidations and can be checked in NuSMV as follows:

CTL#3: SPEC AG ! (BeamLevel.state = ESet & Interface.state = XBeamReady)

CTL#4: SPEC AG ! (BeamLevel.state = XSet & Interface.state = EBeamReady)

Both of these properties can be proved to be true in NuSMV, i.e. these inconsistent states can never occur.

Other possible feature interactions in the Therac-25 can be analysed based on the classification presented in Chapter 3 as follows.

Multiple Action Interaction: In the Therac-25 software, all commands are synchronised from the Interface component to the different subcomponents of the machine component such as BeamLevel, Spreader and the BeamFire. A multiple action interaction can occur if there is a conflict between the actions of sub-components. To check this type of interaction, the following style of temporal logic formula needs to be checked:

SPEC AG (BeamLevel.state = XSet & Spreader.state = OutOfPlace)

This model checking formula indicates the undesirable situation where the state of the BeamLevel component is XSet and the spreader position is OutOfPlace. This formula is proved to be false, which means that this multiple action interaction never occurs in the system.

Shared Trigger Interaction: In the Therac-25 system, the different components synchronise on the shared events SelectX and SelectE. For this reason, a single event triggers two different components:

SPEC AG (SelectX → AF(BeamLevel.state = XSet & Spreader.state = InPlace)

and similarly,

SPEC AG (SelectE → AF(BeamLevel.state = ESet & Spreader.state = OutOfPlace)

In the original Therac-25 (without fault tolerance), these properties would both have failed. However, with fault tolerance and the added guard condition, it can be shown in NuSMV that these properties are both true.

Sequential Action Interaction: Similarly, another potential feature interaction can be where the Spreader receives the SelectX command and brings the spreader InPlace. The resulting change in the environment may trigger other components to perform some action. However, in the Therac-25 system, no component detects movement of the spreader, so this type of feature interaction will not occur.

Resource Contention: In the original Therac-25 system, the Spreader component receives the XSet command from the Interface component and is in the process of bringing the spreader InPlace, while the BeamLevel component resets the beam mode to ESet. This created a resource contention for the Spreader. However, the introduction of the fault tolerant mechanism (specifically the guard condition) has prevented this interaction occurring.

5.5 Summary

In this chapter, the voting mechanism from the orthogonal fault tolerance framework approach has been applied to the Interface component of the computer-controlled medical machine Therac-25. The N-version programming fault tolerance mechanism has been applied to the Therac-25 components, and composition applied according to the operational semantics presented in Chapter 3. The Lex and Yacc pre-processor processed the fault tolerance generics based on the algorithm presented in Chapter 3. This generates the input language for the NuSMV model checking tool that is used for the feature interaction analysis. CTL/ LTL logic properties are applied to the NuSMV model of the Therac-25 to verify the given properties for the composed components.

In this Chapter, it has been shown that the well documented and published serious software error is removed with the introduction of N-version programming fault tolerance. Further potential feature interactions in the Therac-25 software system were classified and validated using CTL/LTL logic properties in NuSMV model checking.

Further evaluation and analysis, spanning both case studies, will now be presented in Chapter 6.

Chapter 6

Evaluation and Analysis

6.1 Introduction

The goals of this chapter are to evaluate the OFT framework and to demonstrate the successful integration of design diversity fault-tolerance mechanisms with system components. The evaluation of this framework is based on the criteria outlined in Chapter 3: separation of concerns, expressiveness of fault-tolerance mechanism for the orthogonality, composability and compositionality for the fault tolerance composition, and soundness of the feature interaction analysis approach. The evaluation draws on the worked example of Home Automation and the Therac-25 system as presented in Chapters 4 and 5.

The main purpose of the evaluation is to determine whether the approach presented in this thesis meets the desirable criteria described in Chapter 1 and which have formed the comparison criteria in Chapter 2. In particular, the evaluation will determine whether this framework can accurately express the orthogonal view of design diversity fault tolerance mechanisms and support their composition, followed by support for the detection of undesirable interactions. In order to measure these criteria, the results will be compared based on the following sub-criteria:

Separation of fault tolerance concerns: Concerns are defined as primary entities for decomposing software into manageable and comprehensible modules. In this thesis, different features of software design diversity fault tolerance mechanisms are dealt with separately, including consideration of their dependency relationships and constraints. The orthogonal and separate fault tolerance concerns are considered helpful in reducing complexity if they can be demonstrated to be easy to manage, design and modifiable.

Expressiveness of fault tolerance: with the help of the orthogonal feature model and generics, it is possible to describe different features of software fault tolerance mechanisms such as error processing technique, checkpointing, etc. The approach is considered expressive if it describes a wide range of features of fault tolerance easily.

Composability: The composition of fault tolerance features with the system component need to be automatically composable. Hence functional components of the software system cannot interfere with the fault tolerance component before this composition. This is achieved by separation of fault tolerance concerns.

Compositionality: In this thesis, it would be helpful to determine if the composition approach were compositional. To validate this property, the approach would need to satisfy the following:

$$(C_A||F_1) || (C_B||F_2) \equiv (C_A|| C_B) || (F_1||F_2) \qquad \text{(i)}$$

$$(C_A||F_1) || (C_B||F_1) \equiv (C_A|| C_B) || F_1 \qquad \text{(ii)}$$

Where C_A and C_B are components and F_1 and F_2 represent the fault tolerance components.

Soundness: The interaction analysis approach presented in this thesis is considered sound if it is able to detect all the potential interactions arising through the composition of fault tolerance with the system component. The model checking approach with temporal logic is used to analyse the composed behaviour for such undesirable interactions. Note that a sound analysis may give false positives; i.e. in terms of interaction analysis, it may find interactions that are not considered as 'bad' interactions.

6.2 Hypotheses

Given the above evaluation criteria, the following hypotheses are posited, the verification or falsification of which will determine the success or failure of the proposed orthogonal fault tolerance framework in the thesis.

H1: The framework presented in this thesis is considered as **expressive** as it provides separation of concern with an orthogonal view of design diversity fault tolerance mechanisms. For the purpose of this evaluation, generics are applied on two case studies that are able to automatically compose fault tolerance with the system component. Both the system and fault tolerance mechanisms remain independent and orthogonal.

H2: The composition mechanism presented in this thesis is **composable** and **compositional**. For the purpose of this evaluation, the mathematical laws of commutativity and associativity are applied.

H3: The orthogonal fault tolerance framework is **sound** with respect to the interactions found. For the purpose of this evaluation, the NuSMV model checking tool is used and this approach will be considered sound if it is able to detect undesirable interactions that are introduced due to the orthogonal handling of fault tolerance mechanisms.

6.3 Analysis

In this section the evaluation criteria are examined, and conclusions drawn as to the veracity or otherwise of the proposed hypotheses.

Separation of Fault Tolerance Concerns

For both case studies, an orthogonal view of different design diversity fault tolerance mechanism has been presented. Different fault tolerance mechanisms were presented in terms of a feature diagram with dependency relationships and constraints. Furthermore, fault tolerance features were handled independently and orthogonally with the help of generics, and a pre-processor underpinned by operational semantics was used to automatically compose fault tolerance with the underlying system components. In this manner, a good separation of fault tolerance concerns was evident, where this separation enables the complexity of introducing fault tolerance to be managed.

Expressiveness of the Fault Tolerance Framework

The OFT framework is considered expressive in several aspects, including the ability to express a variety of recovery behaviours in response to occurrence of failure. Different features can be composed automatically with the underlying system component. The labelled transition system and operational semantics increase its expressiveness through the addition of guard conditions, actions, and the ability to model features such as multiple versions.

However, a limitation of the OFT framework with respect to fault tolerance expressiveness is that this approach is inapplicable for the real time situations and dynamic nature of today's fault tolerant software systems. To make it applicable to dynamic systems, there is a need to study and explore existing methods to address dynamic configuration of the system such as through the use of domain-specific modelling languages (DSMLs) to model the dynamic architecture and the set of valid system configurations.

Composability

Orthogonal fault tolerance features are automatically composed with the components of Home Automation and the Interface component of the Therac-25 software. In both of these examples, ‘generics’ are used to express the fault tolerance features and the composition is based on the operational semantics on labelled transition systems.

In the Home Automation case study, a new component Climate Control was easily integrated and composed with the system. However, for this component, no fault tolerance mechanism was added, but feature interaction was detected as shown in Chapter 4. The resolution to this feature interaction could be the introduction of a further fault tolerance feature into the composed system, which is discussed further below.

Compositionality

In the Home Automation case study, two independent acceptance test features of a recovery block mechanism are composed with the Light Controller component and the Home Status Controller component. In this case, although the acceptance tests are independent, they are both of the same style of fault tolerance, namely recovery block. Hence, regarding the two compositionality formulae presented above, there is only one fault tolerance mechanism, F_1.

Applying this to the case of the Home Automation case study, the second compositionality formula $(C_A || F_1) || (C_B || F_1) \equiv (C_A || C_B) || F_1$ can be shown to hold as follows:

$$(LC || F) || (HSC || F) \equiv (LC || HSC) || F$$

Regarding the left hand side of this equivalence relation: in Chapter 4, Figures 4.7a and 4.7b represent the composition of the Light Controller with the fault tolerance feature and the Home Status Controller with the fault tolerance feature respectively, as given by the inference rules below.

$$\frac{LC_i \xrightarrow{e} LC_i' , F \xrightarrow{c} F'}{LC_i || F \xrightarrow{c,e} LC_i'} \quad (LC || F)$$

$$\frac{HSC_i \xrightarrow{e} HSC_i' , F \xrightarrow{c} F'}{HSC_i || F \xrightarrow{c,e} HSC_i'} \quad (HSC || F)$$

Both of these composed state machines (i.e. figures 4.7a and b) can themselves be composed using the pre-processor and underlying operational semantics to give the state machine shown in Figure 4.8.

Regarding the right hand side of this equivalence relation: in a similar way, Figure 4.3 represents LC || HSC, as also given by the inference rule:

$$\frac{\mathrm{LC}_i \overset{\mathrm{e}}{\rightarrow} \mathrm{LC}_i' \text{ , } \mathrm{HSC}_i \overset{\mathrm{e}}{\rightarrow} \mathrm{HSC}_i'}{\mathrm{LC}_i \| \mathrm{HSC}_i \overset{\mathrm{e}}{\rightarrow} \mathrm{LC}_i' \| \mathrm{HSC}_i'} \quad (\mathrm{LC} \| \mathrm{HSC})$$

This can itself be composed with F, using the pre-processor, and gives the identical state machine as in Figure 4.8.

$$\frac{\mathrm{LC}_i \overset{\mathrm{e}}{\rightarrow} \mathrm{LC}_i' \text{ , } \mathrm{HSC}_i \overset{\mathrm{e}}{\rightarrow} \mathrm{HSC}_i' \text{, } \mathrm{F} \overset{\mathrm{c}}{\rightarrow} \mathrm{F}'}{\mathrm{LC}_i \| \mathrm{HSC}_i \overset{\mathrm{c,e}}{\rightarrow} \mathrm{LC}_i' \| \mathrm{HSC}_i'} \quad (\mathrm{LC} \| \mathrm{HSC} \| \mathrm{F})$$

On the other hand, the first compositionality formula is not satisfied, as two different fault tolerance mechanisms cannot be composed.

$$(\mathrm{LC} \mid\mid \mathrm{F}_1) \mid\mid (\mathrm{HSC} \mid\mid \mathrm{F}_2) \equiv (\mathrm{LC} \mid\mid \mathrm{HSC}) \mid\mid (\mathrm{F}_1 \mid\mid \mathrm{F}_2)$$

For example if one component is using acceptance test feature and the other is using voting feature, these two features cannot be composed. The reason behind this is the constraints associated with these features; for example, these two features may require different execution schemes.

Soundness

The NuSMV tool was used for both studies and was used to check for undesirable feature interactions. These interactions were first classified and then temporal logic formulae were written in LTL and CTL and were used to model check and validate the different formulae. Table 6.1 shows the number of interactions obtained by NuSMV in the analysis of each case study.

Analysis of the Therac-25 case study yielded precisely the correct interactions; that is, the known and published design faults of the system were detected. In the case of the Home Automation system, the identified interactions corresponded to the classifications of

interaction type that were expected. However, it is still a matter for further research to determine if all possible interactions have been found by the proposed technique.

Case Study	**Model Checking Feature Interactions Analysis**
Home Automation	5
Therac-25	4

Table 6.1 Soundness with respect to interaction detection

From the feature interaction analysis, the following conclusions are drawn:

1. Hypothesis H3 has not been falsified and the breadth of the evaluation gives strong confidence that the feature interaction in proposed framework is indeed sound.
2. The larger Therac-25 case study yielded fewer interactions. However, this is not surprising given the well-studied nature of this case study.
3. For the Home Automation case study, the analysis identified all expected interactions and identified no false positives. However, this approach gives no proof that further interactions do not exist.

6.4 Further Analysis

There are two areas of work that merit further discussion relating to further analysis and evaluation of the proposed approach. The first concerns the choice of underlying operational semantics and the second relates to an opportunity to 'reflexively' apply the orthogonal fault tolerance mechanisms as a means to resolve feature interactions. Both will be discussed in turn below.

Choice of Underlying Operational Semantics

As described in Chapter 4, a fault tolerance mechanism (acceptance test) was introduced with two components, Light Controller and Home Status Controller, of the Home Automation system. The operational semantics presented in Section 3.3.5 were designed as synchronous communication semantics, and hence force synchronisation on the shared events, MD and NMD, when the components are composed together. One advantage of this approach is that this composition avoids the occurrence of inconsistent states that may occur if a failure occurs in one of the components where the failure of the acceptance test prevents a transition on event MD or NMD, but the other component's acceptance test is passed and the component transitions as normal (as presented in Figure 4.9).

However, there is also a limitation with this style of composition: in the case of multiple components with an acceptance test and synchronous communication, in the case of failure of a single acceptance test all of the other components are blocked and cannot continue until the failed acceptance test is passed.

In the field of distributed systems, this type of failure is known as a *partial failure*; in such case it is desirable, even in the presence of a fault in one part of the system, that the rest of the system continues to function correctly. This type of failure would require a weakening in the style of operational semantics currently supported by the OFT framework.

The limitation of strictly synchronous operational semantics has been addressed in the field of distributed systems, by the introduction of the notion of 'named localities', as in the work of [Schmitt and Stefani 2004]. In such work, different localities can exhibit different failure and control semantics. For example, whereas a failure in a wide-area network may prevent sub-networks from being able to communicate with each other, the communication within local sub-networks is not restricted. Hence different semantics operate according to the context of the localities.

Reflexive Feature Interaction Resolution through Orthogonal Fault Tolerance

In the Home Automation case study, the actions of the Climate Control component, i.e. opening windows and blinds, trigger the motion detection sensor and result in a number of feature interactions, as has been shown in Figure 4.10.

A possible solution to resolve these interactions is the introduction of a fault tolerance mechanism in the Climate Control component. Consider the scenario presented in Figure 4.11, where shared trigger interactions and sequential action interactions can occur when the climate control service opens the windows and triggers the motion detection. Since motion detection is shared between different components of Home Automation, the movement caused by the climate control service can also trigger the Light Controller to move into the LBP state, the Home Status Controller to move into the H_occupied state, and also turns the security Alarm on.

Suppose now that a fault tolerance mechanism is applied to the sensing element of the Climate Control component, with the purpose of validating the motion sensing. There are two ways in which this may be achieved. The first would be the application of an acceptance test that checks a condition relating to the style of movement (e.g. duration and/ or location of movement). The second approach would be to consider a system that contains multiple

sensors, where voting could be applied on the values of the different sensors, in order to validate the context of motion detected.

In general, it is believed that 'reflexively' applying a fault tolerance mechanism may serve as a valuable mechanism for resolving certain styles of feature interaction such as shared trigger interactions and sequential action interactions. However, more research is required in order to validate this hypothesis.

6.5 Conclusion

Based on the above discussion, the overall evaluation is shown in Table 6.2.

	Evaluation Criteria	**Home Automation**	**Therac-25**
H1: Orthogonality	Separation of Concerns	High	High
	Fault Tolerance Expressiveness	Medium	Medium
H2: Composition	Composability	High	High
	Compositionality	Medium	N/A
H3: Feature Interaction Analysis	Soundness	High	Medium

Table 6.2: Overall Evaluation

The evaluation in this table draws the following conclusions:

1. The OFT framework addresses the separation of fault tolerance concerns for the home automation and the Therac-25 system. The separation of concern criteria is high as it clearly demonstrates the separation of design diversity fault tolerance mechanisms and provides an orthogonal view without adding complexity to the underlying system. In contrast, the fault tolerance expressiveness is medium for both case studies as it only deals with the design diversity fault tolerance mechanism and design faults. For full expressiveness, other styles of fault tolerance techniques, such as data diversity, should also be considered.
2. The Composability criteria for the OFT framework in both case studies is high, as different fault tolerance mechanism are composed with the specified components of the Home Automation and with the Therac-25 component. The compositionality is medium, as in the Home Automation system only formula (ii) is true. Note that in the Therac-25 system, this is not applicable as only one component, namely the Interface component, had fault tolerance applied.

3. For the feature interaction analysis, soundness is considered high in home automation, as the expected interactions were found in NuSMV. Similarly, in the Therac-25 system, the expected interaction is detected and the known software fault prevented.

The above evaluation shows that the orthogonal fault tolerance framework proposed in this thesis brings remarkable benefits for the incorporation of design diversity fault tolerance mechanism with the underlying system component of two case studies. It also provides a model checking approach for the feature interaction analysis. It also shows that this orthogonal fault tolerance approach is absolutely composable and partially compositional as shown in Section 6.3.

The major benefit of orthogonality is to enhance the separation of fault tolerance concern that promotes the independence and isolation of fault tolerance concerns. These properties further reduce complexity as constraints dependencies and relationships between fault tolerance features are provided in detail. Potential undesirable feature interactions are validated in both of the case studies as classified in different categories.

Therefore, this has shown that the approach detailed in this thesis is a promising approach that meets the desirable criteria put forward in Chapter 1: namely, designing fault tolerance at the architecture level, supporting the separation of fault tolerance concerns as an orthogonal view, constructing generics to represent fault tolerance features, and finally supporting the detection of undesirable feature interactions through model checking.

Chapter 7

Conclusions

7.1 Introduction

In this thesis, an Orthogonal Fault Tolerance (OFT) framework is proposed to give an orthogonal view of different features of design diversity fault tolerance mechanisms. A pre-processor has been developed to automatically compose fault tolerance features with the underlying system components and undesirable feature interactions are analysed with the help of the model checking tool NuSMV. The conclusions of the thesis are presented in this chapter, which is structured as follows: Section 7.2 presents a summary of the thesis on a chapter-by-chapter basis, Section 7.3 then reviews the main research results of the thesis followed by other significant results, Section 7.4 identifies directions for future work, and finally, Section 7.5 concludes the chapter and the thesis with some closing remarks.

7.2 Summary of the thesis

Chapter 1 introduced the background to this thesis, focusing on the need for reliable systems and providing fault tolerance at the early stages of software development, such as at the requirement specification and design phase. Furthermore, it introduced key techniques within fault tolerant systems such as data diversity and design diversity techniques, single version and multiple version mechanisms, and the potential for a separation of concerns approach within this field. The remainder of the chapter then discussed the research aims and objectives, and the contributions and the structure of the thesis.

Chapter 2 then surveyed the existing approaches under three main areas, namely: fault tolerance at the requirement specification and design phase, composition techniques for fault tolerance behaviours, and feature interaction analysis. Different approaches under each area have been discussed and analysed with different characteristics in order to address the challenges associated with the composition of fault tolerance. The analysis showed that no surveyed platform addressed all the requirements in a balanced way.

Chapter 3 presented the proposed approach and framework of this thesis in addressing the challenges identified in Chapter 1 and 2. In particular, an Orthogonal Fault Tolerance (OFT) framework was introduced. Constituents of the OFT were described, namely the feature model for different design diversity fault tolerance mechanisms, and dependency relationships and constraints associated with different features of fault tolerance mechanisms. The Lex and Yacc pre-processor was developed based on the algorithms for the automatic composition of fault tolerance and handling of fault tolerance 'generics'. Finally, feature interaction analysis was described with a help of CTL and LTL logic properties supported by a model checking tool NuSMV.

Chapter 4 illustrated the overall framework with the worked example of a Home Automation system. Two components of a Home Automation system, a Light Controller (LC) and a Home Status Controller (HSC) have been used for illustrative purposes. Different fault tolerance mechanisms were automatically composed with the components of Home Automation, with this composition being based on the operational semantics presented in Chapter 3. The chapter concluded by presenting an analysis of undesirable feature interactions.

Chapter 5 applied the Orthogonal Fault Tolerance framework to the Therac-25 case study, a computer controlled medical machine for radiotherapy treatment. Firstly, serious accidents of the Therac-25 were discussed that occurred due to a software error in the system. Then the existing published formalisms to describe and validate that error were presented. The fault tolerance mechanism from the proposed framework was introduced into the Interface component of the Therac-25. The automatic composition of these fault tolerance mechanisms, with the help of a pre-processor and interaction analysis in NuSMV, was shown to successfully overcome the known problem in software design.

Chapter 6 provided an overall evaluation of the OFT framework. First, the evaluation criteria were presented such as separation of concerns, expressiveness of fault tolerance, composability and compositionality, and soundness of the feature interaction approach. Both case studies Home Automation system and the Therac-25 machine were evaluated against these criteria. Finally the chapter revisited the objectives and challenges presented in chapters 1 and 2, and made a comparison of the OFT framework with other approaches.

7.3 Contributions of the thesis

The main contributions of the thesis are as follows.

1. OFT Framework

This thesis has proposed an Orthogonal Fault Tolerance framework to handle different features of design diversity fault tolerance mechanisms as separate concerns. The separation of fault tolerance concerns was the first main aim of the proposed approach and brought the following benefits:

a) To manage the complexity of the underlying system where fault tolerance concerns were addressed independently.

b) To provide an orthogonal view of fault tolerance concerns relative to the underlying system components. This orthogonal view also shows the dependency relationships and constraints associated with different features of fault tolerance mechanisms.

c) Different features of fault tolerance mechanisms were presented in terms of a feature diagram that is simple and consistent.

Importantly, the thesis has shown that the fault tolerance mechanisms can indeed be usefully separated and dealt with as separate concerns.

2. Fault Tolerance Composition through 'Generics'

In this framework, the concept of 'generics' have been introduced and presented, along with an algorithm and pre-processor tool that automatically composed them with the underlying system components. This composition was the second main aim of the thesis and was underpinned by operational semantics applied to labelled transition systems for both the fault tolerance generics and the components of the base system. The pre-processor automatically transformed the composition to the input language model of NuSMV model checking tool.

This is therefore a successful achievement of the thesis objectives, and opens up further research areas including further developing the composition mechanism and underlying operational semantics into a model that offers different semantics for different contexts of failure within large-scale distributed systems, as discussed in chapter 6.

3. Feature Interaction Analysis through Model Checking

The analysis of undesirable feature interactions was the third main objectives of the thesis. The use of temporal logic and the NuSMV model checking tool to analyse these interactions

have been outlined. First, classification is made for different feature interaction categories and then these classifications are represented with a style of temporal logic in CTL/ LTL in NuSMV model checking tool. The model checker validates the property or provides a counter trace for the desired property. This also opens up a further research area through refining the feature interaction analysis with a 'reflexive' approach to feature interaction resolution, as discussed in chapter 6.

In addition to these main contributions of the thesis, further contributions that have been made by this research are as follows:

4. Research and Identification of the Problem Domain

A comprehensive survey of state of the art techniques and state of practice approaches to software fault tolerance has been given with an abundant bibliography that covers progress in this field. The research and the background study have been used in this thesis as the basis for the development of the proposed framework with its subcomponents. It is believed that it will be helpful for any further research and development by other researchers in the area of design diversity software fault tolerance composition and feature interaction analysis.

5. Platform and Domain Independent Framework

The OFT framework presented in this thesis has not been tied to a particular underlying platform, component model, or domain for its realisation. The proposed framework can be used to introduce design diversity fault tolerance to any platform and to any component model based on any domain.

6. Underlying Operational Semantics for the Composition

Another significant contribution of the thesis has been the underpinning of the automatic fault tolerance composition by operational semantics. The semantics and behaviour of a system have a significant impact on the composition and on the overall system.

However the framework, as presented, has used synchronous operational semantics. As mentioned above, for certain types of complex system this may be viewed as a limitation of the approach.

7.4 Future Work

Two areas of further work have already been considered in chapter 6 and briefly referred to above, namely (i) a study of alternative underlying semantics that move beyond the synchronous composition as presented within the OFT framework, and (ii) an investigation in the further feasibility of the OFT framework offering a 'reflexive' feature interaction resolution approach.

Beyond these two areas, this section describes some further directions for future work that may be carried out based upon the proposed framework presented in this thesis:

Extension for Dynamically Adaptive Systems

The proposed framework, as presented, is only applicable to static configurations of systems. However, dynamically adaptive systems have been increasing in use and popularity, whereby the system's behaviour is changed in response to its operational context, user requirements, or needs of other systems and services with which it interacts. Such systems must adapt in the presence of threats and faults and be able to react to hazardous situations. The proposed framework, if extended for dynamically adaptive system, must be able to provide trustworthiness and dependability to those systems. One direction to explore here is the models@run.time community and their use of software models to support runtime reasoning [Bencomo et al. 2014].

Integration with other Feature Interaction Approaches

The proposed framework for the analysis of feature interactions is based on the model checking of temporal logic properties. Many different styles of feature interaction analysis such as software engineering (including focussed techniques and process models), formal methods, and online techniques have been proposed in the literature that offers different benefits in different circumstances. In reality, it may be that a hybrid approach can be used to more fully support the feature interaction analysis associated with fault tolerant systems.

Further Case Studies

Further work is needed to apply the OFT framework to larger and more complex case studies. There are two possible directions that this work could go. Firstly, systems such as those used in home care for the elderly and smart cities may be explored and examined for the variety of fault tolerance mechanisms that they will exhibit. Secondly, large scale and

complex distributed systems could be studied for different failure models and associated semantics.

7.5 Concluding Remarks

Over the last decades, providing fault tolerance capability at the implementation level has been the traditional way for achieving reliability. Such approaches were time and cost effective. However, modern research has also focused on providing fault tolerance at the requirement specification and design levels, and formal properties relating to reliability can be guaranteed with the help of verification and model checking support. Despite the research undertaken in providing fault tolerance at the early stages of software development, the major short-comings have been the lack of fault tolerance expressiveness in terms of maintaining a separation of concerns, its composition with the underlying system component and analysing the potential feature interactions raised by this composition. This thesis has argued the need for a framework that introduced fault tolerance at the specification and design level while having an independent and orthogonal view. Hence, the Orthogonal Fault Tolerance framework has been presented, whose goal has been to embrace the automatic composition of orthogonal design diversity fault tolerance and the analysis of potential undesirable feature interactions. It is hoped that the thesis has provided a significant contribution to future directions in this field.

References

[Abrial 1996] J.R. Abrial, “Assigning Programs to Meanings”, Cambridge University Press, 1996. ISBN 0-521-49619-5.

[Agarwala and Tanik 1989] S. Agarwala and M.M. Tanik, “System Specification with communicating sequential processes (CSP)”, Technical report (Southern Methodist University, 1989.

[Alhir 1998] S.S. Alhir, “UML in Nutshell: A Desktop Quick Reference”, 1998.

[Amman and Knight 1987] P.E. Amman and J.C. Knight. Data Diversity: An Approach to Software Fault Tolerance. In Proc. of 17th Intl. Symposium on Fault Tolerant Computing, June 1987.

[Anderson and Lee 1981] T. Anderson and P.A. Lee, Fault Tolerance: Principles and Practice, Prentice/Hall, 1981.

[Avizienis 1971] A.A. Avizienis, "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design," IEEE Trans. Comp. vol. C-20, no. 11, November 1971, pp. 1,322-1,331.

[Avizienis 1976] A.A. Avizienis, "Approaches to Computer Reliability: Then and Now," Proc. AFIPS NCC, vol. 45, 1976, pp. 401-411.

[Avizienis and Chen 1977] A.A. Avizienis, L. Chen, “On the implementation of N-version programming for software fault tolerance during execution”. In Proceedings of the IEEE International Computer Software and Applications Conference, 1977, pp. 149-155.

[**Avizienis and Kelly 1984]** A.A. Avizienis, and J.P.J. Kelly, “Fault Tolerance by Design Diversity: Concepts and Experiments,” IEEE Computer, Vol. 17, No. 8, 1984.

[Avizienis 1985] A.A. Avizienis, “The N-Version Approach to Fault-Tolerant Software” December 1985, pp. 290 - 300.

[Baniassad and Clarke 2004] E. Baniassad, and S. Clarke. "Theme: An approach for aspect-oriented analysis and design." In Proceedings of the 26th International Conference on Software Engineering, pp. 158-167. IEEE Computer Society, 2004.

[Bencomo et al. 2014] N. Bencomo, R.B. France, H.C. Cheng, U. Aßmann. "Models@run.time - Foundations, Applications, and Roadmaps [Dagstuhl Seminar 11481, November 27 - December 2, 2011]. Lecture Notes in Computer Science 8378, Springer 2014, ISBN 978-3-319-08914-0.

[Blom et al. 1994] J. Blom, B. Jonsson, and L. Kempe. Using temporal logic for modular specification of telephone services. In L.G. Bouma and H. Velthuijsen, editors, [15], pp. 197–216, May 1994.

[Bolton et al. 2008] M.L. Bolton, E.J. Bass, and R.I. Siminiceanu. "Using formal methods to predict human error and system failures." In Proc. 2nd Int. Conf. Appl. Human Factors Ergonom. 2008.

[Bolognesi et al. 1987] T. Bolognesi, and E. Brinksma. "Introduction to the ISO specification language LOTOS." Computer Networks and ISDN systems 14, no. 1, 1987, pp. 25-59.

[Bredereke 2000] J. Bredereke. Families of formal requirements in telephone switching. In Feature Interactions in Telecommunications and Software Systems VI, IOS Press, pp. 257–273, May 2000.

[Brito et al. 2005] P.H.S. Brito, and C.M.F. Rubira. "A framework for analyzing exception flow in software architectures." ACM SIGSOFT Software Engineering Notes 30, no. 4, 2005, pp. 1-7

[Brito et al. 2009] P.H.S. Brito, R. deLemos, C.M.F. Rubira, and E. Martins. "Architecting fault tolerance with exception handling: verification and validation." Journal of Computer Science and Technology 24, no. 2, 2009, pp. 212-237.

[Brito et al. 2009] P.H.S. Brito, C.M.F. Rubira and R. deLemos, "Verifying architectural variabilities in software fault tolerance techniques, 2009.

[Calder et al. 2002] M. Calder, M. Kolberg, E.H. Magill and S. Reiff-Marganiec, "Feature Interaction: A Critical Review and Considered Forecast", 2002.

[**Cain 1992]** M. Cain. Managing run-time interactions between call processing features. In IEEE Communications Magazine, pp. 44–50, February 1992.

[Chen and Avizienis 1978] L. Chen, and A. Avizienis, "N-Version Programming: A fault Tolerance approach to Reliability of software Operation," University of California Los Angeles, 1978.

[**Chitchyan et al 2007]** R. Chitchyan, A. Rashid, P. Rayson, and R.W. Waters, “Semantics-based Composition for Aspect-Oriented Requirements Engineering”, 2007, pp. 36-48.

[**Chitchyan et al. 2005]** R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M.P. Alarcon, J. Bakker, B. Tekinerdogan, S. Clarke, and A. Jackson. "Survey of aspect-oriented analysis and design approaches", 2005.

[Chou 1997] T.C.K. Chou, Beyond Fault Tolerance, IEEE Computer, April 1997, pp. 47–49.

[Clarke, Grumberg and Peled 1999] E.M. Clarke, O. Grumberg, and D.A. Peled, Model checking. Cambridge MA: MIT Press, 1999.

[Clarke and Walker 2001] S. Clarke and R.J. Walker. Composition patterns: An approach to designing reusable aspects. In The 23rd International Conference on Software Engineering (ICSE), Toronto, Canada, 2001.

[Clarke and Walker 2002] S. Clarke and R.J. Walker. Towards a standard design language for AOSD. In The 1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, April 2002.

[Clifton and Leavens 2002] C. Clifton and G.T. Leavens, “Observers and assistants: A proposal for modular aspect-oriented reasoning”, 2002.

[Cottenier et al. 2007] T. Cottenier, A.V.D. Berg, and T. Elrad, "The Motorola WEAVR: Model weaving in a large industrial context." Aspect-Oriented Software Development (AOSD), Vancouver, Canada 32 (2007): 44.

[Cottenier, Berg and Elrad 2006] T. Cottenier, A. V. D. Berg, and T. Elrad, “Modeling Aspect-Oriented Compositions”, 2006.

[Dahll and Lahti 1979] G. Dahll and J. Lahti. An investigation into the methods of production and verification of highly reliable software, 1979.

[Daniel and Ruben 2005] L. Daniel and A. Ruben: Formal Verification of Fault Tolerance Aspects, 2005.

[deLemos 2007] R. deLemos, C. Gacek and A. Romonovsky, “Architecting Dependable Systems”, volume 2677, 2007.

[deLemos 2001] R. deLemos, Describing evolving dependable systems using co-operative software architectures. In ICSM ’01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM’01), 2001.

[Dijkstra 1982] E.W. Dijkstra, "On the role of scientific thought". Selected writings on Computing: A Personal Perspective. New York, NY, USA: Springer-Verlag. pp. 60–66. ISBN 0-387-90652-5.

[Elmendorf 1972] X.R. Elmendorf, "Fault-Tolerant Programming ," Proc. 1972 Int. Symp. Fault-Tolerant Computing, June 1972, pp. 79-83.

[Felty and Namjoshi 2000] A. Felty and K. Namjoshi. Feature specification and automatic conflict detection. pp. 179–192, May 2000.

[Filho Guerra and Rubira 2003] F.C. Filho, P.A.C. Guerra, and C.M.F. Rubira, “An Architectural-Level Exception Handling System for Component-Based Applications”. In LADC, 2003.

[Filho, Brito and Rubira 2006] F.C. Filho, P.H.S. Brito, and C.M.F. Rubira, “Specification of Exception Flow in Software Architectures” - Special Issue on Architecting Dependable Systems, 2006.

[Filho et al. 2005] F.C. Filho, C.M.F. Rubira, and A. Garcia. "A quantitative study on the aspectization of exception handling." In ECOOP Workshop, p. 137, 2005.

[Fleurey et al. 2008] F. Fleurey, B. Baudry, R. France, and S. Ghosh. "A generic approach for automatic model composition." In Models in Software Engineering, pp. 7-15. Springer Berlin Heidelberg, 2008.

[France et al. 2004] R. France, I. Ray, G. Georg, and S. Ghosh. "Aspect-oriented approach to early design modelling." IEE Proceedings-Software 151, no. 4, 2004, pp. 173-185.

[Goldman and Katz 2007] M. Goldman and S. Katz, “Modular aspect verification”, pp. 308-322, 2007.

[Guelfi et al. 2004] N. Guelfi, R. Razavi, A. Romanovsky, and S. Vandenbergh, “DRIP Catalyst: An MDE/MDA Method for Fault-tolerant Distributed Software Families Development”. In OOPSLA and GPCE, 2004.

[Hay and Atlee 2000] J. Hay and J.M. Atlee. Composing Features and Resolving Interactions. pp. 110–119, 2000.

[Hecht 1979] H. Hecht, "Fault-tolerant software for real-time applications," ACM Computing Surveys, vol. 8, no. 4, pp.391-407, 1976.

[Horning et al. 1974] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith and B. Randell, "A program structure for error detection and recovery," Lecture Notes in Computer Science, vol. 16, pp.177-193, 1974.

[Issarny and Banatre 2001] V. Issarny, and J. Banatre, Architecture-based exception handling. In HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences, 2001.

[Jacky 1999] J. Jacky, Lessons from the formal development of a radiation therapy machine control program. In Michael G. Hinchey and Jonathan P. Bowen, editors, Industrial-Strength Formal Methods in Practice, pp. 185–206. Springer-Verlag, 1999.

[Katz et al 2008] E. Katz, S. Katz, W. Havinga, T. Staijen, N. Weston, F. Tainai, A. Rashid and H. Nguten, "Detecting interference among aspects", 2008.

[Keck and Kuehn 1998] D.O. Keck and J. Kuehn, "The feature and service interaction problem in telecommunications systems: A survey." Software Engineering, IEEE Transactions on 24, no. 10, 1998, pp. 779-796.

[Kelly and Avizienis 1983] J.P.J. Kelly and A. Avizienis. A specification-oriented multi-version software experiment, 1983.

[Kelly et al. 1995] B. Kelly, M. Crowther, J. King, R. Masson, and J. DeLapeyre. Service validation and testing. In Feature Interactions in Telecommunications Systems III, pp. 173–184, October 1995.

[Kimbler and Sobirk 1994] K. Kimbler and D. Sobirk. Use case driven analysis of feature interactions. In Feature Interactions in Telecommunications Systems, IOS Press, pp. 167–177, May 1994.

[Kolberg et al. 2003] M. Kolberg, E.H. Magill, and M. Wilson. "Compatibility issues between services supporting networked appliances." Communications Magazine, IEEE 41, no. 11, 2003, pp. 136-147.

[Krishnamurthi et al. 2004] S. Krishnamurthi, K. Fisher and M. Greenberg, "Verifying aspect advice modularly", 2004.

[Kulkarni et al. 2005] S.S. Kulkarni, B. Bonakdarpour, and A. Ebnenasir. "Mechanical verification of automatic synthesis of fault-tolerant programs." In Logic Based Program Synthesis and Transformation, pp. 36-52. Springer Berlin Heidelberg, 2005.

[Laibinis and Troubitsyna 2004] L. Laibinis, and E. Troubitsyna, "Fault Tolerance in a Layered Architecture: A General Specification Pattern in B". In Software Engineering and Formal Methods (SEFM '04).

[Lanfang et al. 2012] T. Lanfang, T. Qingping, X. Jianjun and Z. Huiping, Formal Verification of Signature-monitoring Mechanisms by Model Checking, ComSIS, Vol. 9, No. 4, 2012.

[Laprie 1990] J. Laprie, "Definition and Analysis of Hardware and Software Fault Tolerance, 1990.

[Leveson 1993] N. Leveson, An Investigation of the Therac-25 Accidents. IEEE Computer, Vol. 26, No. 7, July 1993, pp. 18-41.

[Leveson 1995] N. Leveson, "Medical devices: The Therac-25." Appendix of: Safeware: System Safety and Computers, 1995.

[Leveson and Turner 1993] N. Leveson and C.S. Turner. An investigation of the Therac-25 accidents. IEEE Computer, 26(7), pp. 18–41, July 1993.

[Magee and Maibaum 2006] J. Magee and T. Maibaum, "Towards specification, modelling and analysis of fault tolerance in self managed systems". In SEAMS 2006.

[Marples and Magill 1998] D. Marples, and E.H. Magill. "The Use of Rollback to Prevent Incorrect Operation of Features in Intelligent Network Based Systems." In FIW, pp. 115-134. 1998.

[McMillan 1993] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993

[Meng, Anita and Daniel 2009] Z. Meng, L. Anita and J.S. Daniel, "Analyzing Formal Verification and Testing Efforts of Different Fault Tolerance Mechanisms", 2009.

[Mine] Mine Pump Control System, http://www.ic.unicamp.br/˜ra014861/FaTC2 [accessed 31/5/16].

[Mustafiz and Kienzle 2009] S. Mustafiz, and J. Kienzle. "DREP: A requirements engineering process for dependable reactive systems." In Methods, Models and Tools for Fault Tolerance, pp. 220-250. Springer Berlin Heidelberg, 2009.

[Nakamura et al. 2000] M. Nakamura, T. Kikuno, J. Hassine, and L. Logrippo. "Feature Interaction Filtering with Use Case Maps at Requirements Stage." In FIW, pp. 163-178, 2000.

[NuSMV] New Symbolic Model Checker, http://nusmv.fbk.eu/ [accessed 31/5/16].

[Owre et al. 1996] S. Owre, J. Rushby, N. Shankar, S. Rajan & M.K. Srivas, “PVS: Combining Specification, Proof Checking, and Model Checking”, 1996.

[Parchas and deLemos 2004] E. Parchas and R. deLemos, An Architectural Approach for Improving Availability in Web Services. In Third Int. Workshop on Architectures for Dependable Systems, WADS 2004.

[Plath and Ryan 2000] M. Plath and M. Ryan. Defining Features for CSP: Reflections on the Feature Interaction Contest. pp. 202–216, 2000.

[Pohl et al. 2006] K. Pohl, F.V. Linden, A. Metzger: Software Product Line Variability Management. SPLC, 2006: 219

[Pradhan 1996] D.K. Pradhan, Fault-Tolerant Computer System Design, Prentice-Hall, Inc., 1996.

[ProB] ProB Modelling Language, https://www3.hhu.de/stups/prob/ [accessed 31/5/16].

[Randell 1975] B. Randell, “System Structure for Software Fault Tolerance,” IEEE Transactions on Software Engineering, 1975, University of Newcastle UK.

[Reiff 2000] S. Reiff. Identifying resolution choices for an online feature manager. In Feature Interactions in Telecommunications and Software Systems VI, IOS Press, pp. 113–128, May 2000.

[Romonovsky 2007] A. Romanovsky, “On exceptions, exception handling, requirements and software lifecycle,” in 2007 Proceedings of the 10th IEEE International Symposium on High Assurance Systems Engineering, Nov. 2007, pp. 301–301.

[Rubira et al. 2005] C.M.F. Rubira, R. deLemos, G. Rodrigues M. Ferreira, and F.C. Filho. "Exception handling in the development of dependable component-based systems." Software: Practice and Experience 35, no. 3, 2005, pp. 195-236.

[Schauerhuber et al. 2007] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer, and G. Kappel. "A survey on aspect-oriented modeling approaches." Relatorio tecnico, Vienna University of Technology, 2007.

[Schmitt and Stefani 2004] A. Schmitt and J. Stefani, The Kell calculus: A family of higher-order distributed process calculi. In Lecture Notes in Computer Science, Springer-Verlag. Workshop of Global Computing, 2004.

[Scott 1983] R. Scott, ''Data Domain Modelling of Fault-Tolerant Software Reliability,'' Ph.D. Thesis, North Carolina State University at Raleigh, 1983.

[Soares et al. 2012] C. Soares, R.S. Moreira, R. Morla, J. Torres, and P. Sobral. "Prognostic of feature interactions between independently developed pervasive systems." In Prognostics and Health Management (PHM), pp. 1-8, IEEE, 2012.

[Thomas 1993] M. Thomas, "The Story of the Therac-25 in LOTOS", 1993. http://www.dcs.gla.ac.uk/~muffy/papers/HIS1.ps [accessed 31/5/16].

[Tomoyuki, Tatsuhiro and Tohru 2001] Y. Tomoyuki, T. Tatsuhiro, K. Tohru" Automatic verification of Fault Tolerance Using Model Checking", 2001.

[Torres 2000] P.W. Torres, "Software fault tolerance: A tutorial", NASA/TM-2000-210616, 2000.

[Turner 1993] C.S. Turner, "An Investigation of the Therac-25 Accidents." COMPUTER 18, no. 9I62/93 (1993): 0700-001830300.

[Turner 2000] K.J. Turner, "Realising architectural feature descriptions using LOTOS. "CALCULATEURS PARALLELES RESEAUX ET SYSTEMES REPARTIS 12, no. 2, 2000: pp. 145-188.

[UPPAAL] Integrated Modelling Tool, http://www.uppaal.org [accessed 31/5/16].

[Welch and Martin 2000] P. Welch, J. Martin. "Formal Analysis of Concurrent Java Systems" Communicating Process Architectures, 2000.

[Weston et al. 2007] N. Weston, T. Francois, and A. Rashid. "Interaction analysis for fault-tolerance in aspect-oriented programming", 2007, pp. 95-102.

[Yeung and Schneider, 2003] W.L. Yeung and S.A. Schneider: Formal Verification of Fault Tolerant Software Design. The CSP Approach, 2003.

[Zhang et al. 2007] J. Zhang, F. Yang, and S.U. Sen. "Detecting feature interactions in web services with model checking techniques." The Journal of China Universities of Posts and Telecommunications 14, no. 3, 2007, pp. 108-112.

Appendix A

Pre-processor and NuSMV - The complete implementation

Home Automation

A.1 Purpose of Lex and Yacc

The following section A.2 depicts the Lex file for the components of the Home Automation such as Light Controller and the Home Status Controller.

A.2 The Scanner: LC.flex

```
/*********************Scanner for the Finite State Machines Language**************************/

%{
/*=====================================================================
C-Libraries and Token Definitions
=====================================================================*/
#include <stdio.h>
#include <string.h>
#include "y.tab.h"                   /* Token definitions and yylval*/
void yyerror(char *);
%}
/============Token Definitions and Regular Expressions and Rules for FSM===/
%%
 [\t\f\r\n ]+              ;
component                  return COMPONENT;
states                     return STATES;
start                      return START;
events                     return EVENTS;
transitions                return TRANSITIONS;
[a-zA-Z][a-zA-Z0-9_]*      { yylval.str = strdup(yytext); return WORD; }
[0-9]*                     {yylval.str = strdup(yytext); return NUMBER; }
"="                        return EQUALS;
":"                        return COLON;
"{"                        return LCB;
"}"                        return RCB;
","                        return COMMA;
";"                        return SEMICOLON;
.                                  { fprintf( stderr,"unexpected char '%c'!\n", yytext[0]);
                           /* or yyerror("unexpected char '%c'!\n"); */
                           exit(-1);
                           }
%%
/================================END of Rules=====================/
 /*==================== subroutines ============================== */
int yywrap()
{         return 1; }
/*
void yyerror(char *str)
{       fprintf(stderr,"%s\n",str);
}
int main()
{
          //======== open a file handle to a particular file:===================
          FILE *fh = fopen("LCfsm.txt", "r");           // make sure it's valid:
          if (!fh) {
```

```
			yyerror("I can't open LCfsm.txt");
			return -1;
		}		// set lex to read from it instead of defaulting to STDIN:
		yyin = fh;
			// lex through the input:
		yylex();
		return 0;
}
*/
```

A.3 Input Configuration File: LCfsm.txt

The configuration file for the Light Controller and the Home Status Controller is represented in A.3 as follows:

```
/*=========================Configuration File =============================*/
component = LC:
states = {AL_on, LBP };
events = {MD, NMD};
start  = AL_on; MD;
transitions = {
		AL_on, _, MD, _, LBP;
		AL_on, _, NMD, _, AL_on;
		LBP, _, MD, _, LBP;
		LBP, _, NMD, _, AL_on;
		};
component = HSC:
states = {H_empty, H_occupied };
events = {MD, NMD};
start  = H_empty; MD;
transitions = {
		H_empty, _, MD, _, H_occupied;
		H_empty, _, NMD, _, H_empty;
		H_occupied, _, MD, _, H_empty;
		H_occupied, _, NMD, _, H_occupied;
		};

FTconfiguration
Version 3;
AT condition;
Recovery points;
```

A.4 The Parser: LC.yacc

```
%{/*******************Parser for simple finite state machine********************/
/*=========================================================================
				C Libraries, Parser rules and & other C code
=========================================================================*/
#include <stdio.h>		/* For I/O */
#include <stdlib.h>		/* For malloc */
#include <string.h>		/* for strcmp */
#define YYSTYPE char*	/* for string type */
int yylex(void);
const int len = 10;		// max length of state and event names
const int len_t = 10;		// max length of transitions
```

```
const int arraylen = 10;     // max number of states and events
const int arraylen_t = 20;   // max number of transitions
const int len_c = 10;        // max length of conditions (for transitions)
const int len_a = 20;        // max length of actions (for transitions)

char compName[len];
char startState[len];
char statearray[arraylen][len];
char eventarray[arraylen][len];

typedef struct {
   char s1[len];       // start state
   char c[len_c];      // condition
   char e[len];        // event
   char a[len_a];      // action
   char s2[len];       // end state
} Transition;
Transition transitionarray[arraylen_t];
int i = 0;
char pos = 's';    // will hold value s (states), i (initial), e (events) or t (transitions)
void yyerror(char *str);
int isValidState(char *str);
int isValidEvent(char *str);
int isValidTransition(Transition *t);
int isValidCondition(char *cond);
int isValidAction(char *action);
char* getTransition(Transition *t);
//#define DEBUG
%}
// tokens are strings or structs (representing transitions)
%union {
   char *str;
// TO DO: struct for transitions
}
%token COMPONENT EQUALS WORD COLON STATES START EVENTS TRANSITIONS LCB RCB
COMMA SEMICOLON UNDERSCORE
%%
/*=================Start Grammar Rules for finite state machine=================*/
// ---- COMPONENT ----
component :
            COMPONENT EQUALS componentName COLON componentParts
            ;
componentName :
            WORD
            {
#ifdef DEBUG
   printf("#componentName\n");
#endif
            strcpy(compName, yylval);
            // print out component name
printf("COMPONENT:\n  %s\n", compName);
            }
```

```
            ;
componentParts:
            states start events transitions
            ;
// ---- STATES ----
states   : stateLabel EQUALS LCB list RCB SEMICOLON
            {
#ifdef DEBUG
   printf("#states\n");
#endif
            // print out list of states
            printf("STATES:\n");
            int count;
            for (count=0; count<i; count++)
                        printf("  s[%d] = %s\n", count, statearray[count]);
            }
            ;
stateLabel : STATES
            {
#ifdef DEBUG
printf("#stateLabel\n");
#endif
            // new component part so set pos to new value and make sure array index is reset
            pos = 's';
            i = 0;
            }           ;
// ---- START STATE ----
start : startLabel EQUALS startName SEMICOLON
            ;
startLabel : START
            {
#ifdef DEBUG
printf("#startLabel\n");
#endif
            // new component part so set pos to new value and make sure array index is reset
            pos = 'i';
            i = 0;
            }           ;
startName : WORD
   {
#ifdef DEBUG
printf("#startName\n");
#endif
   // check word is in list of states
   if (isValidState(yylval)) {
      // success so print out start state
      strcpy(startState, yylval);
printf("START STATE:\n  %s\n", startState);
      }
   else {
      yyerror("Error: start state not contained in list of states\n");
      exit (-1);
```

```
        }
    }   ;
// ---- EVENTS ----
events  : eventLabel EQUALS LCB list RCB SEMICOLON
            {
#ifdef DEBUG
printf("#events\n");
#endif
            // print out list of events
            printf("EVENTS:\n");
            int count;
            for (count=0; count<i; count++)
                        printf("  e[%d] = %s\n", count, eventarray[count]);
            }           ;
eventLabel : EVENTS
            {
#ifdef DEBUG
printf("#eventLabel\n");
#endif
            // new component part so set pos to new value and make sure array index is reset
            pos = 'e';
            i = 0;
            }
            ;
// ---- LIST OF NAMES ----
list        : name
        | list COMMA name
        ;
name    : WORD
            {
#ifdef DEBUG
printf("#name\n");
#endif
            // we've found a state or event name, so add to appropriate array
            if (pos=='s') {
        strcpy(statearray[i], yylval);
                        }
            else if (pos=='e') {
        strcpy(eventarray[i], yylval);
        }   // increment array counter
            i++;
            }           ;
// ---- TRANSITIONS ----
transitions : transitionLabel EQUALS LCB listTransitions RCB SEMICOLON
            {
#ifdef DEBUG
printf("#transitions\n");
#endif
            // print out list of transitions
            printf("TRANSITIONS:\n");
            int count;
            for (count=0; count<i; count++)
```

```
      printf("  t[%d] = %s\n", count, getTransition(&transitionarray[count]));
      }     ;
transitionLabel : TRANSITIONS
            {
#ifdef DEBUG
printf("#transitionLabel\n");
#endif
            // new component part so set pos to new value and make sure array index is reset
            pos = 't';
            i = 0;
            }
            ;
listTransitions    : transition
            | listTransitions SEMICOLON transition
            ;
transition:
            s1name COMMA condition COMMA ename COMMA action COMMA s2name
            {
#ifdef DEBUG
printf("#transitions\n");
#endif
   // increment array counter
   i++;
            }
            ;
// ---- TRANSITION ELEMENTS ----
s1name: WORD
            {
#ifdef DEBUG
printf("#sname\n");
#endif
            // we've found a state name, so check it's valid
            if (!isValidState(yylval) ) {
      yyerror("Error: state not contained in list of states:");
      yyerror(yyval);
                        exit (-1);
                        }
   strcpy(transitionarray[i].s1, yylval);
   }
   ;
condition : UNDERSCORE
            { // still to implement
#ifdef DEBUG
printf("#condition\n");
#endif
   strcpy(transitionarray[i].c, "_");
   }        ;
ename  : WORD
            {
#ifdef DEBUG
printf("#ename\n");
#endif
```

```
            // we've found an event name, so check it's valid
            if (!isValidEvent(yyval) ) {
        yyerror("Error: event not contained in list of events:");
        yyerror(yyval);
                        exit (-1);
                        }
    strcpy(transitionarray[i].e, yylval);
            }
            ;
action   : UNDERSCORE
            { // still to implement
#ifdef DEBUG
printf("#action\n");
#endif
    strcpy(transitionarray[i].a, "_");
    }
            ;
s2name : WORD
    {
#ifdef DEBUG
printf("#sname\n");
#endif
    // we've found a state name, so check it's valid
    if (!isValidState(yylval) ) {
        yyerror("Error: state not contained in list of states:");
        yyerror(yyval);
        exit (-1);
        }
    strcpy(transitionarray[i].s2, yylval);
    }
    ;
// ---- END RULES ----
%%
extern FILE *yyin;
extern int yylex();
extern int yyparse();
int isValidState(char *str)
{
    int count;
    for (count=0; count<arraylen; count++) {
        //printf("str is -%s-; array element %d is -%s\n",str,count,statearray[count]);
        if (strcmp(str, statearray[count]) == 0)
                                    return 1;
        }
    return 0;
}
int isValidEvent(char *str)
{
    int count;
    for (count=0; count<arraylen; count++) {
        //printf("str is -%s-; array element %d is -%s\n",str,count,statearray[count]);
                        if (strcmp(str, eventarray[count]) == 0)
```

```
                    return 1;
        }
return 0;
}
int isValidTransition(Transition *t)
{
        if (isValidState(t->s1)
                    && isValidCondition(t->c)
                    && isValidEvent(t->e)
                    && isValidAction(t->a)
                    && isValidState(t->s2) )
                return 1;
        else
                return 0;
}
int isValidCondition(char *cond)
{       //still to implement
        return 1;
}
int isValidAction(char *action)
{       //still to implement
        return 1;
}
char *getTransition(Transition *t)
{   char *trans;
   trans = (char *)malloc(len_t);
   strcpy(trans, t->s1);
   strcat(trans, ", ");
   strcat(trans, t->c);
   strcat(trans, ", ");
   strcat(trans, t->e);
   strcat(trans, ", ");
   strcat(trans, t->a);
   strcat(trans, ", ");
   strcat(trans, t->s2);
   //printf("printTransition returning %s\n", trans);
   return trans;
}
void yyerror(char *str)
{
     fprintf(stderr,"%s\n",str);
}
/*==============================Main===================================*/
int main()
{
        // open a file handle to a particular file:
        FILE *fh = fopen("LCfsm.txt", "r");
        // make sure it's valid:
        if (!fh) {
     yyerror("Error: I can't open LCfsm.txt");
                return -1;
        }
```

```
        // set lex to read from it instead of defaulting to STDIN:
        yyin = fh;
        do {
        yyparse();
        } while (!feof(yyin));
        return 0;
}
```

A.5 The compilation sequence:

```
Yacc –d myfile.yacc
Flex myfile.l
cc lex.yy.c y.tab.c –o myfile.exe –lfl
./myfile.exe <config.txt
```

A.6 The Output Files:

Light Controller NuSMV Code

What follows is the input for NuSMV for the Light Controller with the recovery block fault tolerance. This code is automatically generated by Lex and Yacc.

```
MODULE main
VAR
state: {AL_on, LBP};
RP : {AL_on, LBP};
version : 1..3;
-- sensing being examined
sensing : {MD, NMD};
-- true if AT is passed
ATpassed: boolean;
--RP: {None, AL_on, LBP};
ASSIGN
init(state) := AL_on;
next(state) := case
state = AL_on & (sensing = NMD & ATpassed = TRUE) : LBP;
state = LBP & (sensing = MD & ATpassed = TRUE) : AL_on;
state = AL_on & (ATpassed = FALSE) : AL_on ;
state = LBP & (ATpassed = FALSE) : LBP ;
TRUE : {AL_on, LBP};
esac;
ASSIGN
--init(RP) := None;
next(RP) := case
state = AL_on & (ATpassed = FALSE) : AL_on;
state = LBP & (ATpassed = FALSE) : LBP;
TRUE : {AL_on, LBP};
esac;
ASSIGN
init(version) := 1;
next(version) := case
ATpassed = FALSE & version < 3 : version + 1 ;
--version = 3 : 1;
TRUE : {version};
```

```
esac;
/*===========================================================================*/
```

Home Status Controller NuSMV Code

What follows is the input for NuSMV for the Home Status Controller with the recovery block fault tolerance. This code is automatically generated by Lex and Yacc.

```
MODULE main
VAR
state: {H_empty, H_occupied};
RP : {H_empty, H_occupied};
version : 1..3;
-- sensing being examined
sensing : {MD, NMD};
-- true if AT is passed
ATpassed: boolean;
--RP: {None, AL_on, LBP};
ASSIGN
init(state) := H_empty;
next(state) := case
state = H_empty & (sensing = MD & ATpassed = TRUE) : H_occupied;
state = H_occupied & (sensing = NMD & ATpassed = TRUE) : H_empty;
state = H_empty & (ATpassed = FALSE) : H_empty ;
state = H_occupied & (ATpassed = FALSE) : H_occupied ;
TRUE : {{H_empty, H_occupied}};
esac;
ASSIGN
--init(RP) := None;
next(RP) := case
state = H_empty & (ATpassed = FALSE) : H_empty;
state = H_occupied & (ATpassed = FALSE) : H_occupied;
TRUE : {H_empty, H_occupied};
esac;

ASSIGN
init(version) := 1;
next(version) := case
ATpassed = FALSE & version < 3 : version + 1 ;
--version = 3 : 1;
TRUE : {version};
esac;
/*===========================================================================*/
```

Traces of Home Status Controller running in NuSMV

The figure below shows the traces of Home status controller with acceptance test fault tolerance.

```
Command Prompt - nusmv  -int hscat.smv
ATpassed = FALSE
NuSMV > simulate -r -k 3
********  Simulation Starting From State 1.1   ********
NuSMV > show_traces -v
<!-- ################### Trace number: 1 ################### -->
Trace Description: Simulation Trace
Trace Type: Simulation
-> State: 1.1 <-
      state = H_empty
      RP = H_empty
      version = 1
      sensing = NMD
      ATpassed = FALSE
-> State: 1.2 <-
      state = H_empty
      RP = H_empty
      version = 2
      sensing = NMD
      ATpassed = FALSE
-> State: 1.3 <-
      state = H_empty
      RP = H_empty
      version = 3
      sensing = MD
      ATpassed = TRUE
-> State: 1.4 <-
      state = H_occupied
      RP = H_empty
      version = 3
      sensing = MD
      ATpassed = FALSE
NuSMV >
```

A.7 NuSMV model for the Composition of the Light Controller and the Home Status Controller

```
MODULE main
VAR
Lcstate: {AL_on, LBP};
Hscstate: {H_empty, H_occupied};
RP : {AL_on, LBP, H_empty, H_occupied};
version : 1..3;
-- sensing being examined
sensing : {MD, NMD};
-- true if AT is passed
ATpassed1: boolean;
ATpassed2 : boolean;

ASSIGN
init(Hscstate) := H_empty;
next(Hscstate) := case
Hscstate = H_empty & (sensing = MD & ATpassed1 = TRUE & ATpassed2 = TRUE) : H_occupied;
Hscstate = H_occupied & (sensing = NMD & ATpassed1 = TRUE & ATpassed2 = TRUE) :H_empty;
Hscstate = H_empty & (ATpassed1 = FALSE | ATpassed2 = FALSE) : H_empty ;
Hscstate = H_occupied & (ATpassed1 = FALSE | ATpassed2 = FALSE) : H_occupied ;
TRUE : {{H_empty, H_occupied}};
esac;
ASSIGN
init(Lcstate) := AL_on;
next(Lcstate) := case
Lcstate = AL_on & (sensing = MD & ATpassed1 = TRUE & ATpassed2 = TRUE) : LBP;
Lcstate = LBP & (sensing = NMD & ATpassed1 = TRUE & ATpassed2 = TRUE) : AL_on;
Lcstate = AL_on & (ATpassed1 = FALSE | ATpassed2 = FALSE) : AL_on ;
```

```
Lcstate = LBP & (ATpassed1 = FALSE | ATpassed2 = FALSE) : LBP ;
TRUE : {AL_on, LBP};
esac;
ASSIGN
--init(RP) := None;
next(RP) := case
Lcstate = AL_on & (ATpassed1 = FALSE | ATpassed2 = FALSE) : AL_on;
Lcstate = LBP & (ATpassed1 = FALSE | ATpassed2 = FALSE) : LBP;
Hscstate = H_empty & (ATpassed1 = FALSE | ATpassed2 = FALSE) : H_empty;
Hscstate = H_occupied & (ATpassed1 = FALSE | ATpassed2 = FALSE) : H_occupied;
TRUE : {H_empty, H_occupied};
esac;

ASSIGN
init(version) := 1;
next(version) := case
ATpassed1 = FALSE & version < 3 : version + 1 ;
ATpassed2 = FALSE & version < 3 : version + 1;
--version = 3 : 1;
TRUE : {version};
esac;
/*=========================================================================*/
```

A.8 CTL and LTL Properties for Feature Interaction Analysis

What follows are the logical properties written in CTL to check the feature interactions based on the classifications presented in Chapter 4.

```
SPEC AG !(LC.state = AL_on & HSC.state = H_occupied)
SPEC AG!(LC.state = LBP & HSC.state = H_empty)
SPEC AG((LC.state = AL_on &!c_LC) -> AX(LC.state = AL_on))
SPEC AG((LC.state = LBP & !c_LC) -> AX(LC.state = LBP))
/*=========================================================================*/
```

A.9 Introduction of a new component ‘Climate Control’ in Home Automation

```
/*=========================Configuration File=============================*/
component = CC:
states = {wind_open, wind_clos };
temp: 18..21;
start  = wind_open; 18;
transitions = {
        wind_open, _, temp <18, _, wind_close;
        wind_open, _, 18<temp<21 _, wind_open;
        wind_close, _, temp>21, _, wind_close;
        wind_close, _, 18<temp<21, _, wind_close;
        };
/*=========================================================================*/
```

Checking Logic Properties for Feature Interaction

a) SPEC AG !((ClimateControl.state = Window_open & MD) -> AF(Alarm_on))

b) SPEC AG (MD -> !AF(Alarm.state = on & (LC.state = LBP & HSC.state = H_occupied)))

Appendix B

Therac-25 – Additional Information

B.1 Input Configuration File: Therac.txt

The configuration file for the Interface component of the Therac-25 is as follows:

```
/*=========================Configuration File ===================================*/

component = Therac_Interface:
state : {Edit, XDataEntry, EDataEntry, XBeamReady, EBeamReady, Treatment_Ad};
events = {XSelect, ESelect, Fire, Reset, change, enter};
start  = Edit;
transitions = {
                Edit, _, XSelect, _, XDataEntry;
                 Edit, _, ESelect, _, EDataEntry;
                XDataEntry , enter, XBeamReady;
                XDataEntry, change,Edit;
                EDataEntry, enter, EBeamReady;
                EDataEntry, change, Edit;
                 XBeamReady, change, XDataEntry;
                EBeamReady, change, EDataEntry;
                XBeamReady, Fire, Treatment_Ad;
                 EBeamReady, Fire, Treatment_Ad;
                Treatment_Ad, Reset, Edit;}

/*==============================End Config File================================/
```

B.2 The Output File: Therac-25 NuSMV Code

The pre-processor generates the following output file (suitable as the input file for NuSMV) based on the specification of the Interface component. This component is further composed with other components of the Therac-25 such as Beam Level, Spreader and the Beam Fire.

```
MODULE T1(command) --//Interface
VAR
        state : {Edit, XDataEntry, EDataEntry, XBeamReady, EBeamReady, Treatment_Ad};
        ASSIGN
        init(state) := Edit;
        next(state) := case
                state = Edit & command = XSelect : XDataEntry;
                state = Edit & command = ESelect : EDataEntry;

                state = XDataEntry & command = enter: XBeamReady;
                state = XDataEntry & command = change: Edit;

                state = EDataEntry & command = enter: EBeamReady;
                state = EDataEntry & command = change: Edit;

                state = XBeamReady & command = change:XDataEntry;
```

```
                state = EBeamReady & command = change:EDataEntry;

                state = XBeamReady & command = Fire: Treatment_Ad;
                state = EBeamReady & command = Fire: Treatment_Ad;
                state = Treatment_Ad & command = Reset : Edit;
        TRUE : state;
        esac;

MODULE T2(command) --//BeamLevel
VAR
                state : {Neither, XSet, ESet};
                Time: 0..8;
                --//event : {Fire, Reset};
        ASSIGN
        init(state) := Neither;
        next(state) := case
                state = Neither &  command = XSelect  : XSet;
                state = Neither &  command = ESelect  : ESet;
                state = XSet &  command = ESelect & Time >=8 : ESet;
                state = ESet  & command = ESelect & Time >=8: XSet;
                state = XSet &  command = ESelect & Time <8 : XSet;
                state = ESet  & command = ESelect & Time <8: ESet;
                TRUE : state;
        esac;

MODULE T3(command) --//Spreader//
VAR
        state : {OutOfPlace, InPlace};
        ASSIGN
        init(state) := OutOfPlace;
        next(state) := case

                state = OutOfPlace & command = XSelect : InPlace;
                state = InPlace & command = ESelect : OutOfPlace;

        TRUE : state;
        esac;
MODULE T4(command) --//BeamFire//
VAR
        state : {waiting, Fired};

        ASSIGN
        init(state) := waiting;
        next(state) := case

                state = waiting & command = Fire : Fired;
                state = Fired & command = Reset : waiting;

        TRUE : state;
        esac;
MODULE main
        VAR
                command : {XSelect, ESelect, Fire, Reset, change, enter};

                Interface : T1(command);
                BeamLevel :T2(command);
                Spreader: T3(command);
```

```
            BeamFire: T4(command);
```

/===/

B.3 Interface Component with N-version Programming Fault Tolerance

```
MODULE T1(command) --//Interface
VAR
            state : {Edit, XDataEntry, EDataEntry, XBeamReady, EBeamReady, Treatment_Ad};
      ASSIGN
      init(state) := Edit;
      next(state) := case
            state = Edit & command = XSelect : XDataEntry;
            state = Edit & command = ESelect : EDataEntry;
            state = XDataEntry & command = enter : XBeamReady;
            state = XDataEntry & command = change: Edit;
            state = EDataEntry & command = enter : EBeamReady;
            state = EDataEntry & command = change: Edit;
            state = XBeamReady & command = change:XDataEntry;
            state = EBeamReady & command = change:EDataEntry;
            state = XBeamReady & command = Fire: Treatment_Ad;
            state = EBeamReady & command= Fire: Treatment_Ad;
            state = Treatment_Ad & command = Reset : Edit;
      TRUE : state;
      esac;

MODULE T2(command) --//BeamLevel
VAR
            state : {Neither, XSet, ESet};
            Time: 0..8;
            --//event : {Fire, Reset};
      ASSIGN
      init(state) := Neither;
      next(state) := case
            state = Neither &  command = XSelect  : XSet;
            state = Neither &  command = ESelect  : ESet;
            state = XSet &  command = ESelect & Time >=8 : ESet;
            state = ESet  & command = ESelect & Time >=8: XSet;
            state = XSet &  command = ESelect & Time <8 : XSet;
            state = ESet  & command = ESelect & Time <8: ESet;
            TRUE : state;
      esac;

MODULE T3(command) --//Spreader//
VAR
      state : {inplace, outofplace};
      ASSIGN
      init(state) := outofplace;
      next(state) := case
      state = outofplace & command = XSelect : inplace;
      state = inplace & command = ESelect : outofplace;
      TRUE : state;
      esac;

MODULE T4(command) --//BeamFire//
VAR
```

```
        state : {waiting, Fired};
        ASSIGN
        init(state) := waiting;
        next(state) := case
                state = waiting & command = Fire : Fired;
                state = Fired & command = Reset : waiting;

        TRUE : state;
        esac;
MODULE main
        VAR
                command : {XSelect, ESelect, Fire, Reset, change, enter};
                Rf : {7,0};
        ASSIGN
                --init(command) :=
                next(command) := case
                Rf = 7 : XSelect;
                Rf = 0 : ESelect;
                TRUE : {XSelect, ESelect};
        esac;
                init(Rf) := 0;
                next(Rf) := case
                Rf = 0 & command = XSelect : 7;
                Rf = 7 & command = ESelect : 0;
                TRUE : Rf;
        esac;
        VAR
                Interface : T1(command);
                BeamLevel :T2(command);
                Spreader: T3(command);
                BeamFire: T4(command);
```

/==/

B.4 CTL and LTL Properties for Feature Interaction Analysis

CTL Properties to check the safe condition and feature interaction analysis for the Therac-25 software:

```
--Safety Property

SPEC AG !(BeamLevel.state = XSet & Spreader.state = OutOfPlace & BeamFire.state = Fired)
SPEC AG ! (BeamLevel.state = ESet & Interface.state = EBeamReady & Spreader.state = InPlace &
BeamFire.state = Fired)
SPEC AG ! (BeamLevel.state = ESet & Interface.state = XBeamReady)
SPEC AG ! (BeamLevel.state = XSet & Interface.state = EBeamReady)

SPEC AG (BeamLevel.state = XSet & Spreader.state = OutOfPlace)
SPEC AG (SelectX → AF(BeamLevel.state = XSet & Spreader.state = InPlace)

SPEC AG (SelectE → AF(BeamLevel.state = ESet & Spreader.state = OutOfPlace)
```

/==/

B.5 Trace of the Interface Component

The output shows that the logical properties are true as expected.